ESCUELA SUPERIOR DE INGENIERÍA

Ingeniería Técnica en Informática de Sistemas

Operadores de mutación de cobertura para WS-BPEL 2.0

Curso 2010-2011

Valentín Liñeiro Barea

Cádiz, 16 de septiembre de 2011

ESCUELA SUPERIOR DE INGENIERÍA

Ingeniería Técnica en Informática de Sistemas

Operadores de mutación de cobertura para WS-BPEL 2.0

DEPARTAMENTO: Lenguajes y Sistemas Informáticos.

DIRECTORES DEL PROYECTO: Antonia Estero Botaro y Juan Boubeta Puig.

AUTOR DEL PROYECTO: Valentín Liñeiro Barea.

Cádiz, 16 de septiembre de 2011

Fdo.: Valentín Liñeiro Barea

Índice general

Índice de figuras

Índice de tablas

1. Introducción

Este proyecto fin de carrera (PFC, de ahora en adelante), ha sido desarrollado por el alumno para colaborar en las investigaciones realizadas por el grupo de investigación "UCASE de Ingeniería del Software", en el seno de la Universidad de Cádiz. La actividad del grupo UCASE comprende actualmente las áreas de *Ingeniería de Servicios*, *Arquitecturas Dirigidas por Eventos*, *Arquitecturas Orientadas a Servicios*, *Desarrollo Dirigido por Modelos*, *Prueba de Software* y *Verificación y Validación de Software*.

Este PFC está orientado a la prueba de composiciones de servicios web (WS). El lenguaje empleado en las composiciones empleadas por el grupo de investigación es WS-BPEL 2.0 [36]. El lenguaje WS-BPEL tiene la ventaja de estar completamente expresado en XML, lo que hace que sea portable a cualquier motor estándar existente. Los WS permiten el desarrollo de aplicaciones distribuidas de manera rápida, simple y con coste bajo, motivo por el cual están cobrando protagonismo a la hora de definir procesos de negocio. Por lo tanto, la prueba de este tipo de software se torna esencial.

La técnica de prueba de software empleada por el grupo es la prueba de mutaciones, un tipo de prueba estructural basada en errores. En líneas generales, la técnica consiste en generar a partir de un programa una serie de programas, los cuales poseen únicamente una diferencia[1] respecto al original. Estos programas se denominan *mutantes*. Para generar los mutantes, se necesita una serie de reglas definidas, los *operadores de mutación*. Se suelen definir dos tipos de operadores de mutación, los que modelan fallos cometidos por los programadores y los que fuerzan una serie de criterios de cobertura.

El grupo de investigación UCASE ha desarrollado la herramienta GAmera [19]. Se trata de una herramienta que permite aplicar la prueba de mutaciones a composiciones WS-BPEL reduciendo el coste computacional que esta produce. Esta reducción del coste computacional se consigue generando solamente un subconjunto de los mutantes totales que puede producir una composición. El subconjunto de mutantes generado está guiado por un algoritmo genético.

La herramienta MuBPEL [5] forma parte de GAmera. Su trabajo consiste en el análisis de las composiciones WS-BPEL, la generación de los mutantes y su posterior ejecución. MuBPEL está orientada a la mutación tradicional, mientras que GAmera está orientada a la mutación evolutiva [18].

[1] Este tipo de mutación es de orden uno. Cabe la posibilidad de realizar más de un cambio sintáctico al programa original, obteniendo un mutante de orden superior.

La herramienta MuBPEL incorpora un conjunto de 26 operadores de mutación que solo tiene en cuenta los fallos sintácticos que pueden cometer los programadores [21]. Este PFC viene a completar ese conjunto de operadores con otros nuevos que permitan aplicar una serie de criterios de cobertura de código y otros relacionados con ellos.

Este PFC se propuso para cubrir la necesidad del grupo UCASE de un conjunto de operadores de mutación capaces de aplicar una serie de criterios de cobertura de código y relacionados con la cobertura de código. El conjunto de operadores disponibles antes de la propuesta de este PFC sólo contemplaban los fallos sintácticos cometidos por los programadores, por lo que este PFC aporta una nueva categoría de operadores de mutación, la de cobertura.

Una de las claves de este PFC es la escasez de bibliografía existente sobre el lenguaje WS-BPEL. Esto hace que los ejemplos de composiciones WS-BPEL existentes sea muy reducido. Esto convierte a este PFC en un proyecto de investigación, ya que será necesario un alto conocimiento del lenguaje para poder obtener un ejemplo al que se le puedan aplicar todos los operadores propuestos.

Este PFC también abarca el estudio de las equivalencias de los operadores de mutación de cobertura para el lenguaje WS-BPEL 2.0 con los operadores de cobertura para otros lenguajes estructurados, como C, Ada o Fortran.

1.1. Pruebas de software

Las pruebas de software [41] consisten en verificar el comportamiento del software a partir de una serie de casos de prueba. Dado que la prueba de cada secuencia dentro del programa es imposible, ya que el número de posibles entradas es infinito, es necesario seleccionar dentro del dominio de entradas un subconjunto de ellas, las cuales conformarán el conjunto de casos de prueba.

Existen principalmente dos enfoques de prueba de software. Uno de ellos consiste en probar que el software funciona correctamente. Sin embargo, según Myers [35], la prueba de software consiste en *"el proceso de ejecutar un programa con la intención de encontrar errores"*. Esto aporta el segundo enfoque, probar que el software tiene errores. Este enfoque es el adecuado a la hora de realizar las pruebas, pues según Dijkstra [16], *"las pruebas sólo pueden demostrar la presencia de errores, no su ausencia"*.

Niveles de prueba de software

A continuación se presentan los principales niveles de prueba de software [41]. Los niveles se muestran en modo ascendente, es decir, desde los módulos hasta la prueba del sistema final.

Prueba de unidad Consiste en probar la lógica de cada módulo del software. Los resultados de las mismas pueden ser: *Éxito*, cuando se obtienen los resultados esperados; *Fracaso*, cuando los resultados obtenidos difieren de los esperados y *Error*, cuando los resultados obtenidos no se ajustan al dominio.

Prueba de integración Consiste en probar la integración de los módulos teniendo en cuenta la estructura del sistema. La integración puede ser:

Incremental Las pruebas se realizan integrando los módulos uno a uno progresivamente.

No incremental Las pruebas se realizan integrando todos los módulos a la vez.

Prueba de sistema Consiste en probar la integración entre software, hardware y usuario.

Prueba de aceptación Consiste en la prueba por parte del usuario en la que decide si el producto se ajusta a los requisitos especificados.

Tipos de prueba de software

Estáticas / dinámicas Las pruebas estáticas se realizan verificando el código del software, y las pruebas dinámicas se realizan ejecutando el software.

De caja blanca / negra Las pruebas de caja blanca están orientadas a la estructura interna del software, y las pruebas de caja negra están orientadas a la especificación del software, es decir, a sus posibles entradas y salidas.

1.1.1. Pruebas de mutaciones

La prueba de mutaciones [28] es una técnica de prueba de software basada en errores. El criterio en que se basa esta técnica es el de medir la calidad de un conjunto de casos de prueba, lo que se denomina análisis de mutaciones. En la figura 1.1 extraída de [28], podemos ver el proceso de análisis de mutaciones.

Sea *P* el programa original. A partir de él, se genera un conjunto *P'* de programas, denominados *mutantes*. Los mutantes son programas que contienen una única diferencia respecto al programa original. Se generan aplicando una serie de reglas predefinidas, los *operadores de mutación*, al código fuente del programa original.

Los operadores de mutación introducen pequeños cambios sintácticos en el programa original, manteniendo la validez sintáctica del mismo, con el motivo de modelar los errores habituales cometidos por los programadores o forzar la aplicación de diferentes criterios de cobertura de código. Son dependientes del lenguaje de programación, así

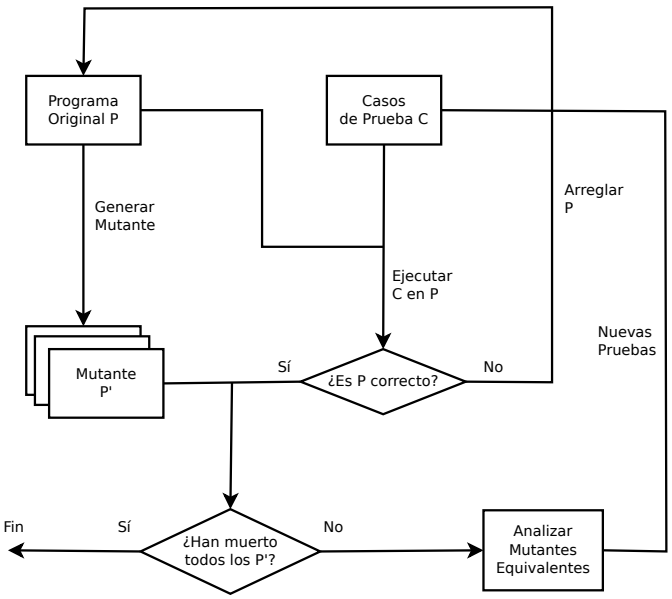

Figura 1.1.: Análisis de mutaciones

pues existen operadores de mutación para C [7], Ada [37], Fortran [31], Java [32], SQL [42], WS-BPEL [21], etc.

Un ejemplo de aplicación de un operador de mutación podría ser el siguiente:

Programa original:

```
while(a - 2 > 0) {
      ...
}
```

Mutante generado tras aplicar el operador EIU, que añade el menos unario:

```
while(-(a - 2) > 0) {
      ...
}
```

El siguiente paso será ejecutar los mutantes sobre el conjunto de casos de prueba del programa original. En función de la salida del mutante, podemos diferenciar 3 estados:

- Si para todos los casos de prueba la salida del mutante coincide con la salida del programa original, el mutante está *vivo*.
- Si para algún caso de prueba la salida del mutante difiere de la salida del programa original, el mutante está *muerto*.
- Si el mutante generado no puede ser ejecutado, el mutante es *erróneo*.

En algunos casos, algunos mutantes vivos siempre poseen la misma salida que el programa original, debido al cambio aplicado. Éstos se denominan mutantes equivalentes. Así pues, tenemos los siguientes tipos de mutantes:

Erróneo Mutante que no puede ser ejecutado.

Muerto Mutante cuya salida difiere de la del programa original para alguno de los casos de prueba.

Vivo Mutante cuya salida coincide con la del programa original para todos los casos de prueba.

Persistente No existen suficientes casos de prueba para matarlo.

Equivalente Provoca siempre la misma salida que el programa original.

Para medir la calidad de un conjunto de casos de prueba es necesario calcular la puntuación de mutación, que consiste en el cociente entre el número de mutantes muertos y el número de mutantes no equivalentes.

El problema que posee el empleo de esta técnica de prueba del software es el coste computacional que conlleva el proceso, ya que existen multitud de operadores de mutación que, aplicados al programa original, generan una gran cantidad de mutantes, los cuales deben ser ejecutados hasta encontrar un caso de prueba que los mate. Por lo tanto, existen una serie de técnicas que ayudan a reducir el coste computacional, como las siguientes:

Técnicas para reducir el número de mutantes

Mutación aleatoria Consiste en ejecutar un subconjunto de mutantes seleccionados aleatoriamente.

Agrupación de mutantes Consiste en agrupar los mutantes en torno a los casos de prueba que los matan, y seleccionar algunos mutantes de cada grupo.

Mutación selectiva Consiste en aplicar un subconjunto de los operadores de mutación disponibles.

Mutación evolutiva Consiste en la ejecución de un subconjunto de mutantes seleccionados mediante un algoritmo genético.

Mutación de orden superior Consiste en aplicar más de un operador de mutación por cada mutante.

Técnicas para reducir el coste de la ejecución de los mutantes

Mutación fuerte Se compara la salida del mutante con la del programa original tras finalizar la ejecución.

Mutación débil Se compara el estado del mutante una vez ejecutado el punto donde reside la mutación.

Optimización del tiempo de ejecución Son técnicas que se apoyan en la optimización aportada por compiladores, intérpretes o metamutantes.

Empleo de plataformas avanzadas Se basan en la aplicación del paralelismo a la hora de ejecutar los mutantes en un cluster, equipos multiprocesadores, redes de ordenadores, etc.

1.1.2. Criterios de cobertura

Los criterios de cobertura [35] son un conjunto de reglas que imponen una serie de requisitos a un conjunto de casos de prueba dado, como, por ejemplo, la ejecución de todas las sentencias o el paso por todas las ramas. Los principales criterios de cobertura de código son los siguientes:

1.1.2.1. Cobertura de sentencias

El criterio de cobertura de sentencias permite obtener un conjunto de casos de prueba que ejecuten todas las sentencias de un programa.

Es un criterio débil, ya que no permite encontrar errores relacionados con las decisiones tomadas en el programa o los caminos de ejecución del mismo.

En la figura 1.2 podemos ver un ejemplo de aplicación de este criterio. Vemos que el conjunto de casos de prueba dado ejecuta todas las sentencias, sin embargo, no detectamos errores como:

- En la primera decisión, debería haber un *or* en vez de un *and*.
- La segunda decisión debería ser $b = 3$ en vez de $b = 2$.
- Existen entradas en las que no se ejecuta ninguna sentencia $(a = 5,\ b < 2)$.

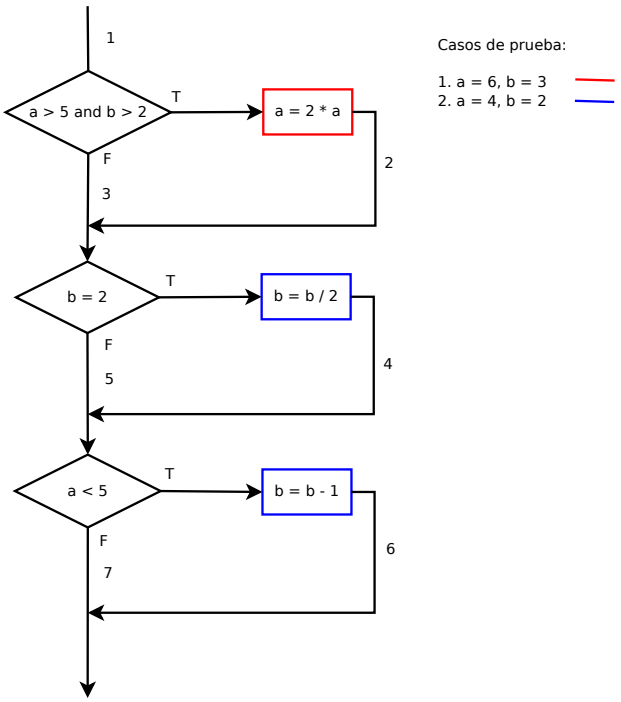

Figura 1.2.: Ejemplo de cobertura de sentencias

1.1.2.2. Cobertura de decisión

El criterio de cobertura de decisión tiene por objetivo generar casos de prueba en los que cada decisión del programa sea evaluada a verdadero y falso, respectivamente.

Es un criterio más fuerte que el anterior, aunque no es aplicable en programas en los que no haya decisiones.

En la figura 1.3 podemos ver un ejemplo de aplicación de este criterio. Vemos que se ejecutan todas las sentencias y todas las decisiones del programa toman todos los valores posibles. De todas formas, este criterio tiene ciertas carencias, ya que en el caso de que la última decisión fuese $a < 4$, no seríamos capaces de detectar este error.

1.1.2.3. Cobertura de condición

El criterio de cobertura de condición se encarga de obtener casos de prueba en los que cada decisión dentro de cada condición tome todos los valores posibles, es decir, verdadero y falso.

En la figura 1.4 vemos la principal debilidad de este criterio. Los casos de prueba especificados cumplen el criterio, ya que todas las condiciones del programa son evaluadas a verdadero y falso, respectivamente. Sin embargo, este conjunto de casos de prueba deja de cumplir el criterio de cobertura de decisión y de sentencias, ya que existe una

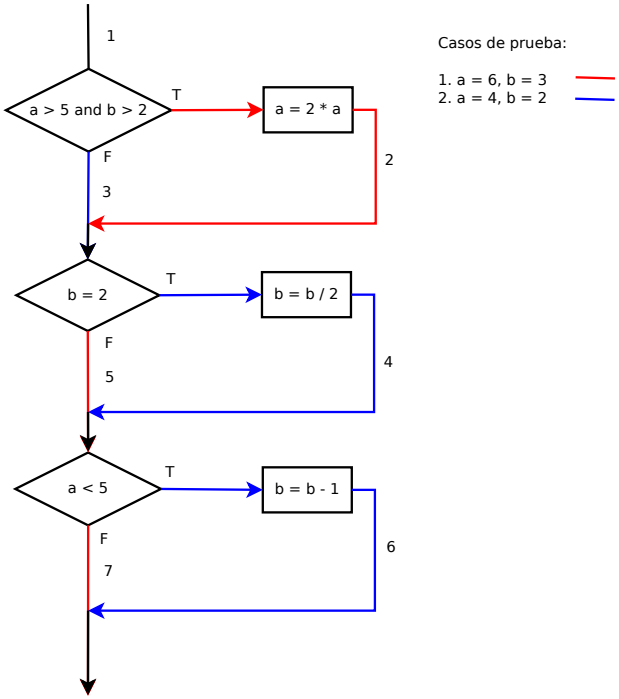

Figura 1.3.: Ejemplo de cobertura de decisión

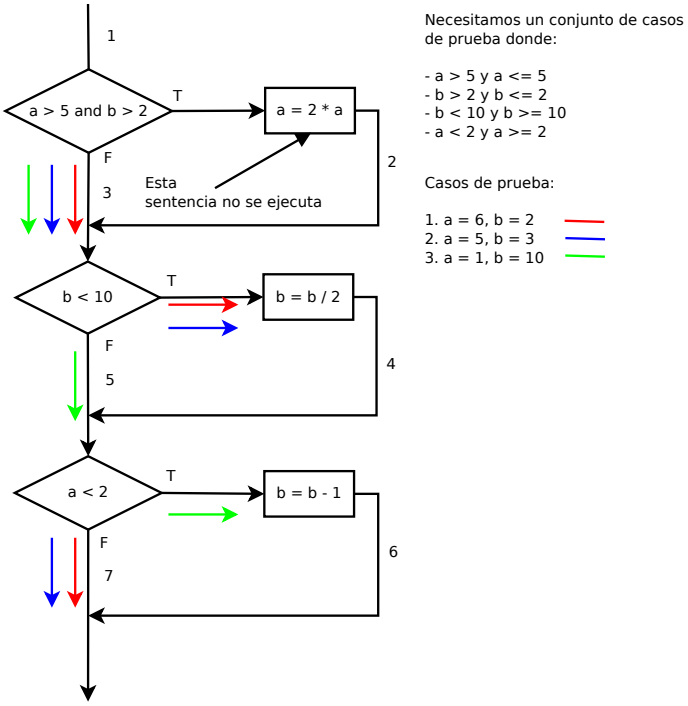

Figura 1.4.: Ejemplo de cobertura de condición

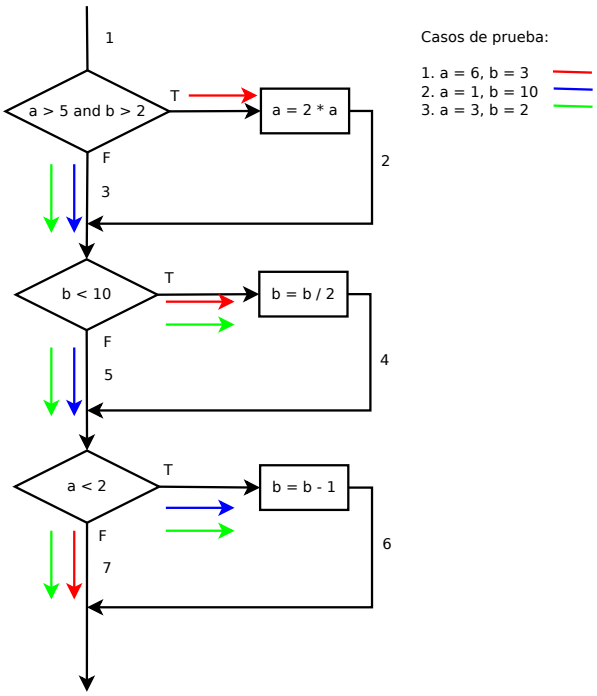

Figura 1.5.: Ejemplo de cobertura de decisión / condición

decisión que no toma su valor verdadero, lo que implica que una sentencia del programa no se ejecute. En resumen, la aplicación de este criterio de cobertura NO garantiza el cumplimiento del criterio de cobertura de decisión.

1.1.2.4. Cobertura de decisión / condición

El criterio de cobertura de decisión / condición es un criterio que combina los dos criterios anteriores, es decir, genera un conjunto de casos de prueba en los que cada decisión y cada condición sean evaluadas a verdadero y falso.

Este criterio viene a paliar las deficiencias que posee el criterio de cobertura de condición, evitando casos como el del ejemplo de anterior, como podemos ver en 1.5. Este criterio subsume a los criterios de cobertura de decisión y de condición. Aunque pueda parecer que mediante este criterio se obtiene cobertura múltiple, esto puede no ser así, ya que en multitud de ocasiones, unas condiciones enmascaran a otras.

1.1.2.5. Cobertura múltiple

El criterio de cobertura múltiple tiene por objetivo ejercitar la tabla de verdad completa de cada decisión preteneciente al programa, es decir, genera casos de prueba en los que

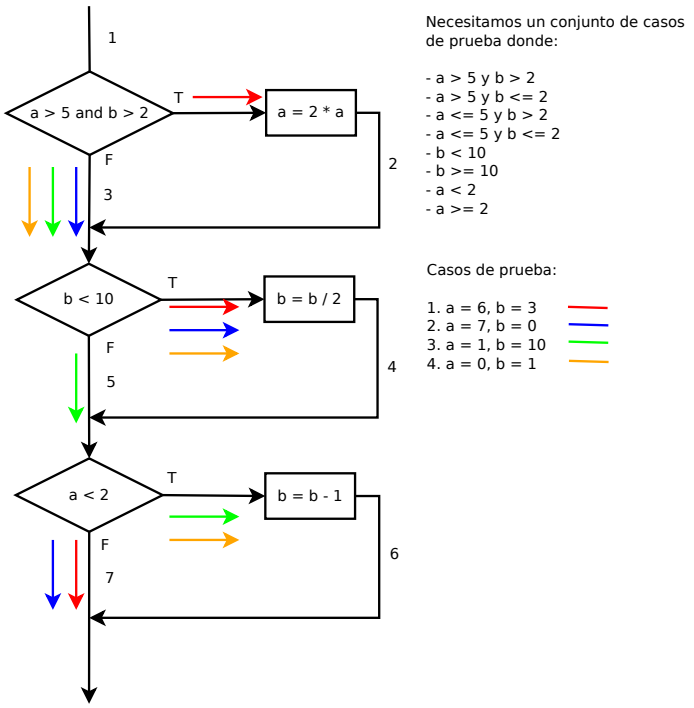

Figura 1.6.: Ejemplo de cobertura múltiple

se dan todas las combinaciones posibles de valores en las condiciones que pertenezcan a la decisión en cuestión.

En la figura 1.6 tenemos un ejemplo en el que se aplica este criterio. Obtenemos un conjunto de casos de prueba en el que se dan todas las combinaciones de valores dentro de cada decisión. Este criterio subsume a todo criterio de cobertura, garantizando cobertura de sentencias, decisión y condición.

En la figura 1.7 podemos qué criterios de cobertura engloban a otros.

1.2. El lenguaje WS-BPEL 2.0

El lenguaje WS-BPEL 2.0 es un lenguaje estandarizado por OASIS [36], que permite definir procesos de negocio a partir de la composición de WS. Está completamente basado en XML [11], cualidad que lo hace portable e independiente del motor donde se ejecute.

Un proceso WS-BPEL se divide en las siguientes secciones:

1. Definición de relaciones con los socios externos, es decir, el cliente y los WS involucrados en el proceso.

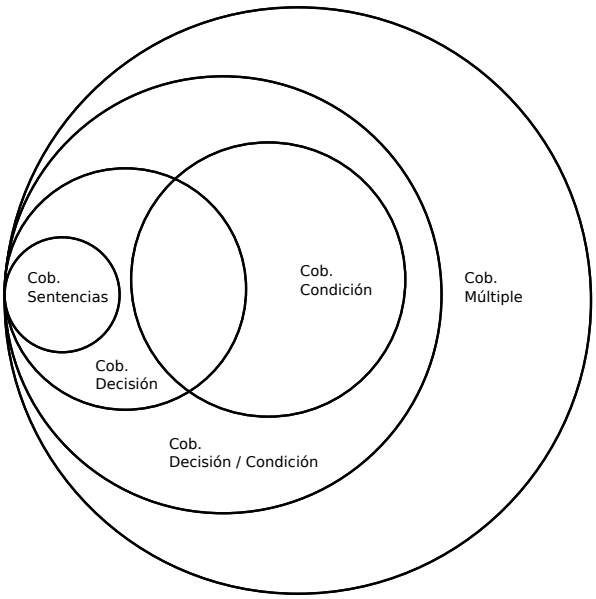

Figura 1.7.: Visión general de los criterios de cobertura

2. Definición de las variables empleadas por el proceso, empleando las tecnologías WSDL [13] y XML Schema (XSD) [9].

3. Definición de los distintos tipos de manejadores que puede utilizar el proceso:

   Manejadores de fallo Contienen la lógica necesaria para gestionar los fallos ocurridos durante la ejecución del proceso o la de un WS involucrado.

   Manejadores de eventos Contienen la acciones necesarias a la hora de recibir peticiones durante la ejecución del proceso.

   Manejadores de terminación Indican las acciones necesarias para terminar el proceso ordenadamente.

   Manejadores de compensación Deshacen la invocación de un WS.

4. Descripción del proceso de negocio mediante las actividades que proporciona el lenguaje.

Los elementos anterirores son todos globales por defecto. Cabe la posibilidad de declarar estos elementos de forma local, empleando el contenedor `scope`. A través de este elemento, se puede dividir el proceso de negocio en distintos ámbitos.

Un proceso WS-BPEL está formado por actividades. Una actividad está representada por un elemento XML, y se le pueden asociar una serie de atributos y un conjunto de contenedores, los cuales pueden poseer atributos asociados. Las actividades pueden ser de los siguientes tipos:

Básicas Son actividades que tienen una labor determinada dentro del proceso de negocio (recepción de mensajes, invocación de servicios, respuesta al cliente, etc.)

Estructuradas Son actividades compuestas por actividades básicas, y son las encargadas de definir la lógica de negocio.

El lenguaje WS-BPEL tiene soporte nativo para la concurrencia, mediante la actividad `flow`. Esta actividad permite la ejecución concurrente de un conjunto de actividades, indicando las condiciones de sincronización necesarias. Un ejemplo de esta actividad es el siguiente:

```
<flow> ← Actividad estructurada
 <links> ← Contenedor
   <link name="comprobarVuelo-A-reservarVuelo" ← Atributo /> ← Elemento
 </links>
 <invoke name="comprobarVuelo" ... > ← Actividad básica
  <sources> ← Contenedor
   <source linkName="comprobarVuelo-A-reservarVuelo" ← Atributo /> ← Elemento
  </sources>
 </invoke>
 <invoke name="comprobarHotel" ... />
 <invoke name="comprobarAlquilerCoche" ... />
 <invoke name="reservarVuelo" ... >
  <targets> ← Contenedor
   <target linkName="comprobarVuelo-A-reservarVuelo" /> ← Elemento
  </targets>
 </invoke>
</flow>
```

Se puede apreciar que la actividad `flow` invoca a tres WS de manera concurrente, `comprobarVuelo`, `comprobarHotel` y `comprobarAlquilerCoche`. El WS `reservarVuelo` sólo se invocará si termina el servicio comprobarVuelo. Tenemos un ejemplo de sincronización de actividades, pues se ha establecido un enlace (`link`) entre ellas. De esta manera, la actividad objetivo del enlace se ejecuta sólo si la actividad fuente se completa.

WS-BPEL soporta distintos lenguajes de expresiones. Todos los motores WS-BPEL estándar existentes soportan el lenguaje de expresiones XPath 1.1 [8]. EL lenguaje XPath es un lenguaje declarativo que permite la realización de consultas en documentos XML. XPath dispone de operadores aritméticos, lógicos y relacionales con una sintaxis muy similar a la de los lenguajes tradicionales.

1.3. Objetivos

Este PFC tiene como objetivo la definición e implementación de un conjunto de operadores de mutación para WS-BPEL 2.0. Estos operadores los podemos clasificar en dos grupos:

Operadores que permiten aplicar criterios de cobertura:

CFA Permite aplicar el criterio de cobertura de sentencias.

CDE Permite aplicar el criterio de cobertura de decisión.

CCO Permite aplicar el criterio de cobertura de condición.

CDC Permite aplicar el criterio de cobertura de decisión / condición.

Operadores que en circunstancias especiales pueden servir para aplicar algún criterio de cobertura:

EIU Añade el menos unario a una expresión aritmética.

EIN Añade la negación a una expresión lógica.

EAP Añade el valor absoluto positivo a una expresión aritmética.

EAN Añade el valor absoluto negativo a una expresión lógica.

Todos los operadores presentados ayudarán a completar el conjunto de operadores definido por Estero et al. [21]. Los operadores que forman parte de ese conjunto son de 4 categorías: mutación de identificadores, mutación de expresiones, mutación de actividades y mutación de condiciones excepcionales y eventos. Con el trabajo realizado en este PFC, se incluye la categoría de cobertura, además de enriquecer la categoría de mutación de expresiones.

Para llevar a cabo el objetivo principal es necesario:

1. Estudiar las similitudes y diferencias de los operadores de cobertura y relacionados para WS-BPEL 2.0 con los de otros lenguajes de programación, ampliando la comparativa realizada en [10].

2. Implementar los operadores presentados, empleando hojas de estilo XSLT. Estas hojas de estilo, una por cada operador, tendrán el código necesario para aplicar las mutaciones especificadas.

3. Crear un conjunto de casos de pruebas unitarias, empleando el framework JUnit [39]. Estos casos de prueba servirán para comprobar que los operadores de mutación realizan correctamente su trabajo.

4. Disponer de una composición WS-BPEL a la que aplicar los operadores implementados. En caso de que no sea así, habrá que adaptar una existente para que se le puedan aplicar los nuevos operadores.

5. Crear un conjunto de casos de prueba que mate a todos los mutantes no equivalentes generados por los nuevos operadores para la composición WS-BPEL elegida o adaptada.

1.4. Alcance

Los productos generados por este PFC son los siguientes:

- Un estudio sobre la definición de operadores de cobertura y relacionados con la cobertura para WS-BPEL 2.0, en el que además se comparan con los existentes para los lenguajes estructurados C, Ada y Fortran.
- El conjunto de hojas XSLT que implementa a los operadores de mutación.
- Las clases Java que implementan las pruebas unitarias a los operadores, empleando JUnit.
- La composición WS-BPEL *LoanApproval* adaptada para la aplicación de estos operadores.
- El fichero *.bpts* con el conjunto de casos de prueba que mata a los mutantes no equivalentes generados por la composición anterior.

1.5. Visión general

El documento comienza con la definición de los operadores de cobertura y relacionados con la cobertura para WS-BPEL. Luego, se realiza una comparativa con los operadores de cobertura para otros lenguajes como C, Ada o Fortran, provenientes del paradigma estructurado.

Más adelante, se comenta el calendario seguido para realizar este PFC y se realiza una breve descripción sobre las restricciones y tecnologías empleadas.

El resto de la memoria se dedica a detallar el proceso de ingeniería seguido, es decir, las fases de análisis, diseño, implementación y pruebas. Se adjuntan al final los manuales de usuario, desarrollador e instalación.

1.6. Glosario

1.6.1. Acrónimos

AST Abstract Syntax Tree

DOM Document Object Model

GUI Graphical User Interface

IDE Integrated Development Environment

SCV Sistema de Control de Versiones

SOAP Simple Object Access Protocol

UML Unified Modeling Language

URL Uniform Resource Locator

XML eXtensible Markup Language

XPath XML Path Language

XSLT eXtensible Stylesheet Language Transformations

WS Web Services

XSD XML Schema Definition

WS-BPEL Web Services Business Process Execution Language

WSDL Web Services Description Language

1.6.2. Definiciones

Condición En un programa, una condición es una expresión lógica atómica (habitualmente, una comparación) que forma parte de una decisión del mismo.

BPR Extensión del fichero que contiene todos los elementos necesarios para el despliegue de una composición WS-BPEL.

Decisión En un programa, una decisión es un punto del mismo donde el flujo de instrucciones se bifurca. El camino a tomar se determina mediante una expresión lógica, la cual es un conjunto de una o más condiciones, conectadas mediante $\wedge$ y $\vee$.

Desplegar En el mundo de los WS, desplegar un proceso consiste en ponerlo en funcionamiento en un motor, donde queda a la espera de las peticiones de los clientes en un puerto especificado.

Mockup Sustituto de un servicio web real empleado a la hora de probar la ejecución de composiciones WS-BPEL con BPELUnit. Se emplea para evitar la utilización de servicios web reales, por motivos de precio o de control del proceso de prueba. Se encargan de responder empleando valores predefinidos, especificados previamente.

PDD Extensión del fichero empleado por el motor ActiveBPEL que contiene la información necesaria para el despliegue de un proceso WS-BPEL.

2. Operadores de mutación de cobertura para WS-BPEL 2.0

En este capítulo se define un conjunto de operadores mutación de cobertura para el lenguaje WS-BPEL 2.0. Además, se definen una serie de operadores de mutación para WS-BPEL 2.0 que en circunstancias especiales pueden servir para aplicar algún criterio de cobertura en concreto. Por último, se estudian las equivalencias existentes entre los operadores definidos para WS-BPEL y los operadores definidos para los lenguajes C, Ada y Fortran.

2.1. Definición de operadores de cobertura para WS-BPEL 2.0

En esta sección se proponen y definen unos operadores de mutación de cobertura para WS-BPEL, cuyas descripciones se resumen en la Tabla 2.1. Estos operadores completarán el conjunto de operadores de mutación definidos por Estero-Botaro et al. [21] que modelan los fallos cometidos por los programadores al escribir código WS-BPEL.

2.1.1. Cobertura de sentencias

CFA es el operador de cobertura de sentencias (ver 1.1.2.1) definido para WS-BPEL. Este operador sustituye una actividad por la actividad `exit`, provocando la salida abrupta de la misma. Veamos a continuación un ejemplo:

Operador	Descripción
CFA	Sustituye una actividad por la actividad `exit`.
CDE	Sustituye una decisión por `true()` o `false()`.
CCO	Sustituye una condición por `true()` o `false()`.
CDC	Sustituye una decisión o condición por `true()` o `false()`.

Tabla 2.1.: Operadores de mutación de cobertura para WS-BPEL 2.0

Programa original:

```
<sequence>
  <if>
    <condition>
    ($a > 1 and $b = 0)
    </condition>
    ...
  </if>
</sequence>
```

CFA-01:

```
<sequence>
  <exit>
</sequence>
```

Como podemos ver en el mutante que se ha generado (denominado CFA-01) al aplicar una vez el operador CFA al programa original, se ha sustituido la actividad `if` por la actividad `exit`, para asegurar la ejecución hasta ese punto. Con esto obtenemos cobertura de sentencias, ya que, generando todos los mutantes, tendremos sustituidas todas las sentencias, y a la hora de ejecutarlos, si todos los mutantes mueren significa que todas las sentencias han sido ejecutadas al menos una vez.

2.1.2. Cobertura de decisión

CDE es el operador de cobertura de decisión que se define para WS-BPEL. Este sustituye cada decisión de la composición por el valor *true* o *false*. Como utilizamos el lenguaje de expresiones XPath, se utilizarán las funciones que proporcionan estos valores: *true()* o *false()*, respectivamente.

Un ejemplo de aplicación del operador CDE es el siguiente:

Programa original:

```
<sequence>
  <if>
    <condition>
    ($a > 1 and $b = 0)
    </condition>
    ...
  </if>
</sequence>
```

CDE-01:

```
<sequence>
  <if>
    <condition>
    true()
    </condition>
    ...
  </if>
</sequence>
```

CDE-02:

```
<sequence>
  <if>
    <condition>
    false()
    </condition>
    ...
  </if>
</sequence>
```

Al igual que ocurría en el caso anterior, tras generar todos los mutantes, tendremos sustituidas todas las decisiones por sus valores posibles. Entonces, si todos los mutantes mueren, significa que todas las decisiones han tomado todos los valores posibles al menos una vez, obteniendo cobertura de decisión.

2.1.3. Cobertura de condición

El operador CCO sustituye cada una de las condiciones existentes en la composición por *true()* y *false()*, es decir, por los valores que pueden tomar. A continuación podemos ver un ejemplo, donde CCO-01, CCO-02, CCO-03 y CC0-04 son los mutantes obtenidos tras aplicar CCO:

Programa original:

```
<sequence>
  <if>
    <condition>
    ($a > 1 and $b = 0)
    </condition>
    ...
  </if>
</sequence>
```

CCO-01:

```
<sequence>
  <if>
    <condition>
    (true() and $b = 0)
    </condition>
    ...
  </if>
</sequence>
```

CCO-02:

```
<sequence>
  <if>
    <condition>
    (false() and $b = 0)
    </condition>
    ...
  </if>
</sequence>
```

CCO-03:

```
<sequence>
  <if>
    <condition>
    ($a > 1 and true())
    </condition>
    ...
  </if>
</sequence>
```

CCO-04:

```
<sequence>
  <if>
    <condition>
    ($a > 1 and false())
    </condition>
    ...
  </if>
</sequence>
```

Cabe destacar que, en composiciones donde las decisiones estén formadas únicamente por una condición, los mutantes generados por este operador son equivalentes a los generados por CDE, por lo que se recomienda aplicar CCO en composiciones donde existan decisiones compuestas por más de una condición.

2.1.4. Cobertura de decisión / condición

El operador CDC cambia todas las decisiones y condiciones del programa por todos los valores que pueden tomar, como establecen los criterios que engloba.

Tomando el mismo programa original presentado anteriormente, este operador producirá los mismos mutantes generados al aplicar el operador de cobertura de decisión (mutantes CDE-01 y CDE-02) y el operador de cobertura de condición (mutantes CCO-01, CCO-02, CCO-03 y CCO-04). Por tanto, la aplicación del operador CDC subsume la de los operadores CCO y CDE.

2.1.5. Cobertura múltiple

Se propuso en la fase de definición de operadores de cobertura un operador relacionado con la cobertura múltiple, denominado CMC. Dicho operador tenía como objetivo la aplicación del criterio de cobertura múltiple, de manera que se generara un conjunto de mutantes que ejercitase la tabla de verdad de cada una de las decisiones existentes en la composición WS-BPEL.

Sin embargo, este efecto se logra con la aplicación del operador CDC, por la naturaleza de las expresiones XPath. Veamos un ejemplo a continuación:

Sea D la decisión:$(a \vee b) \wedge (c \vee d)$, y sean los mutantes:

$M_1 = (\top \vee b) \wedge (c \vee d)$, $M_2 = (\bot \vee b) \wedge (c \vee d)$, $M_3 = (a \vee \top) \wedge (c \vee d)$, $M_4 = (a \vee \bot) \wedge (c \vee d)$, $M_5 = (a \vee b) \wedge (\top \vee d)$, $M_6 = (a \vee b) \wedge (\bot \vee d)$, $M_7 = (a \vee b) \wedge (c \vee \top)$, $M_8 = (a \vee b) \wedge (c \vee \bot)$, $M_9 = \top$ y $M_{10} = \bot$, obtenidos tras aplicar el operador CDC. En la tabla 2.2, se compara la tabla de verdad original con las de las decisiones mutadas, señalando en negrita los valores que difieren, y que por tanto, matan al mutante.

Como podemos ver, los valores que no logra ejercitar la cobertura de condición, los ejercitará la cobertura de decisión, por tanto, se obtiene cobertura múltiple en este caso.

No obstante, es necesario asegurar que este hecho se cumple de manera general, por tanto, queda como trabajo futuro profundizar en este estudio con el objeto de verificar que se cumple para todo tipo de decisiones. En el caso que no sea así, habrá que implementar el operador CMC.

a	b	c	d	D	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}
T	T	T	T	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	**F**
T	T	T	F	*T*	*T*	*T*	*T*	*T*	*T*	**F**	*T*	*T*	*T*	**F**
T	T	F	T	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	**F**	*T*	**F**
T	T	F	F	*F*	*F*	*F*	*F*	*F*	**T**	*F*	**T**	*F*	**T**	*F*
T	F	T	T	*T*	*T*	**F**	*T*	*T*	*T*	*T*	*T*	*T*	*T*	**F**
T	F	T	F	*T*	*T*	**F**	*T*	*T*	*T*	**F**	*T*	*T*	*T*	**F**
T	F	F	T	*T*	*T*	**F**	*T*	*T*	*T*	*T*	*T*	**F**	*T*	**F**
T	F	F	F	*F*	*F*	*F*	*F*	*F*	**T**	*F*	**T**	*F*	**T**	*F*
F	T	T	T	*T*	*T*	*T*	*T*	**F**	*T*	*T*	*T*	*T*	*T*	**F**
F	T	T	F	*T*	*T*	*T*	*T*	**F**	*T*	**F**	*T*	*T*	*T*	**F**
F	T	F	T	*T*	*T*	*T*	*T*	**F**	*T*	*T*	*T*	**F**	*T*	**F**
F	T	F	F	*F*	*F*	*F*	*F*	*F*	**T**	*F*	**T**	*F*	**T**	*F*
F	F	T	T	*F*	**T**	*F*	**T**	*F*	*T*	*F*	*F*	F	**T**	*F*
F	F	T	F	*F*	**T**	*F*	**T**	*F*	*T*	*F*	*F*	*F*	**T**	*F*
F	F	F	T	*F*	**T**	*F*	**T**	*F*	*F*	*F*	*F*	F	**T**	*F*
F	F	F	F	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	**T**	*F*

Tabla 2.2.: Comparación entre la tabla de verdad original y la de los mutantes

Operador	Descripción
EIU	Inserta el menos unario en una expresión aritmética.
EIN	Inserta la negación en una expresión booleana.
EAP	Inserta el valor absoluto positivo en una expresión aritmética.
EAN	Inserta el valor absoluto negativo en una expresión aritmética.

Tabla 2.3.: Operadores de mutación relacionados con la cobertura para WS-BPEL 2.0

2.2. Operadores relacionados con la cobertura para WS-BPEL 2.0

En esta sección se define una serie de operadores de mutación de expresiones que modelan los errores que un usuario podría cometer a la hora de escribirlas (téngase en cuenta que en WS-BPEL las expresiones XPath se suelen escribir a mano, lo que puede ocasionar fallos accidentales en su escritura), continuando con el trabajo definido en [21]. No obstante, a diferencia de los definidos en ese trabajo, los operadores aquí propuestos añaden código WS-BPEL.

Sin embargo, en ciertos contextos pueden servir para que se aplique algún criterio de cobertura, a pesar de que no cumplan ninguno en concreto. Por ejemplo, la adición del menos unario en las expresiones aritméticas puede hacer que el flujo del programa pase o no pase por un conjunto determinado de instrucciones, con lo que se obtiene cierta cobertura de ramas. En la Tabla 2.3 se presenta una breve descripción de los operadores de mutación de expresiones que están relacionados con la cobertura para WS-BPEL 2.0.

2.2.1. Inserción del menos unario

El operador EIU inserta el operador menos unario a una de las expresiones aritméticas de primer nivel, es decir, en el caso de tener una expresión formada por subexpresiones, solo se aplicaría a la expresión completa, no a las anidadas.

Programa original:

```
<assign>
  <copy>
    <from>$i * $i</from>
    <to variable="Resultado"/>
  </copy>
</assign>
```

EIU-01:

```
<assign>
  <copy>
    <from>-($i * $i)</from>
    <to variable="Resultado"/>
  </copy>
</assign>
```

Como puede observarse, se ha añadido el menos unario a la expresión XPath presente en la actividad `from`.

2.2.2. Inserción de la negación lógica

El operador EIN añade el operador de negación a una expresión lógica, sin tener en cuenta el posible anidamiento de la misma.

Aquí tenemos un ejemplo de aplicación:

Programa original:

```
<sequence>
  <if>
    <condition>
    ($a > 1 and $b = 0)
    </condition>
    ...
  </if>
</sequence>
```

EIN-01:

```
<sequence>
  <if>
    <condition>
    not($a > 1 and $b = 0)
    </condition>
    ...
  </if>
</sequence>
```

En este caso, se niega la condición de la actividad `if`.

2.2.3. Inserción del valor absoluto positivo

El operador EAP inserta el valor absoluto[1] positivo a una de las expresiones aritméticas de primer nivel de la composición que se desee mutar, de manera análoga a los operadores anteriores.

A continuación, se presenta un ejemplo de aplicación del operador EAP:

Programa original:

```
<assign>
  <copy>
    <from>$i - $j</from>
    <to variable="Resultado"/>
  </copy>
</assign>
```

[1] En XPath 2.0, el valor absoluto se consigue a través de la función `abs()`. Sin embargo, esta función no está disponible en XPath 1.1, versión que emplea el motor WS-BPEL ActiveBPEL. Es necesario emular su funcionamiento a partir de: $abs(x) = x \cdot ((x \geq 0) - (x < 0))$. Téngase en cuenta que los resultados de las comparaciones se convierten a valor numérico, siendo 1 cuando son verdaderas y 0 cuando son falsas.

EAP-01:

```
<assign>
  <copy>
    <from>
    (($i - $j) * ((($i - $j) >= 0) - (($i - $j) < 0)))
    </from>
    <to variable="Resultado"/>
  </copy>
</assign>
```

En este ejemplo se ha tomado el valor absoluto de la expresión perteneciente al `from` hijo de la actividad `copy`.

2.2.4. Inserción del valor absoluto negativo

El operador EAN inserta el valor absoluto negativo (ver pie 1) a una de las expresiones aritméticas presentes en la composición, excepto en las anidadas.

Veamos la aplicación del operador EAN:

Programa original:

```
<assign>
  <copy>
    <from>$i - $j</from>
    <to variable="Resultado"/>
  </copy>
</assign>
```

EAN-01:

```
<assign>
  <copy>
    <from>
    -(($i - $j) * ((($i - $j) >= 0) - (($i - $j) < 0)))
    </from>
    <to variable="Resultado"/>
  </copy>
</assign>
```

A partir de este programa original, que es el mismo que se tomó para aplicar el valor absoluto positivo, se genera el mutante en el que se ha insertado esta vez el valor absoluto negativo.

2.3. Comparativa de los operadores de cobertura definidos para WS-BPEL y otros lenguajes

En este apartado se realiza una comparativa de los operadores de cobertura definidos para los lenguajes estructurados más populares, como es el caso de C, Ada y Fortran, con los operadores de cobertura definidos para WS-BPEL en la Sección 2.1. Esta comparativa amplía la que se describe en [10], que compara únicamente los operadores que modelan los fallos que pueden cometer los programadores al escribir programas WS-BPEL. Por tanto, completamos el estudio que compara los operadores de mutación definidos para WS-BPEL con los definidos para otros lenguajes, teniéndose en cuenta tanto los operadores que modelan fallos cometidos por los programadores como los que miden ciertos criterios de cobertura.

En la Tabla 2.4 se resume la comparativa de los operadores que hemos definido para WS-BPEL 2.0 en este trabajo con los existentes para otros lenguajes de programación. Los operadores definidos para otros lenguajes que son equivalentes a los definidos para WS-BPEL se han clasificado en dos categorías: operadores de cobertura y los que están relacionados con la cobertura (aunque son operadores que modelan fallos). Todos estos operadores se han separado, a su vez, en dos grupos: los *operadores similares*, aquellos que necesitan ser redefinidos con pequeños cambios sintácticos para que sean equivalentes a los de WS-BPEL, y los *operadores idénticos* (aparecen marcados en negrita en la tabla), que realizan exactamente la misma mutación que los definidos para WS-BPEL.

Como puede observarse en la Tabla 2.4 todos los operadores de mutación de cobertura y los relacionados con la cobertura para Ada son idénticos a los de WS-BPEL, a excepción del operador EIN que no es equivalente a ninguno de Ada.

Sin embargo, ocurre el caso contrario para los operadores definidos para Fortran y C, es decir, todos son similares a los de WS-BPEL, excepto uno de ellos que es idéntico: SAN para Fortran y STRP para C son idénticos a CFA para WS-BPEL.

Los operadores para Fortran similares a los definidos para WS-BPEL son los siguientes. LCR y ROR son equivalentes a ELL y ERR para WS-BPEL, respectivamente, tal y como se comenta en [10]; además, para aplicar los criterios de cobertura de decisión, condición, decisión/condición y múltiple en Fortran, es necesario que sean aplicados conjuntamente, lo que difiere de nuestra definición. El operador UOI es similar a EIU para WS-BPEL porque además de insertar el menos unario también lo elimina. Finalmente, el operador ABS aúna las mutaciones que realizan los operadores de inserción del valor absoluto positivo y negativo en WS-BPEL: EAP y EAN.

No obstante, existen menos operadores de mutación para C que sean similares a los de WS-BPEL. Por un lado, la definición del operador Oior es más completa que la de CDE para WS-BPEL. Por otro lado, Uuor no solo inserta el menos unario como hace EIU para WS-BPEL, sino que además inserta otros operadores unarios como el más o

Categoría	WS-BPEL	Ada	Fortran	C
Cobertura	CFA	**SEE** [37]	**SAN** [31]	**STRP** [7]
	CDE	**CDE** [37]	LCR [31]	Oior [7]
	CCO	**CCO** [37]	ROR [31]	–
	CDC	**CDC** [37]	LCR, ROR	–
Relacionados con la cobertura	EIU	**EUI** [37]	UUI [31]	Uuor [7]
	EIN	–	–	–
	EAP	**EAI** [37]	ABS [31]	VDTR [7]
	EAN	**ENI** [37]	ABS	VDTR

Tabla 2.4.: Equivalencias entre los operadores de cobertura definidos para WS-BPEL 2.0 y otros lenguajes estructurados

el valor absoluto. Finalmente, VDTR es análogo al operador ABS para Fortran y, por tanto, es similar a los operadores EAP y EAN para WS-BPEL.

3. Calendario

En este capítulo se detalla el calendario seguido para llevar a cabo este PFC.

3.1. Fases del proyecto

Toma de contacto con el grupo UCASE

En esta fase, se tomó contacto con el grupo de investigación UCASE.

Se le presentó al alumno una de las líneas de investigación en las que el grupo trabaja actualmente, la línea de *prueba de mutaciones*. Luego, el grupo indicó la necesidad de implementar un conjunto de operadores de cobertura para poder realizar estudios específicos de cobertura en las composiciones WS-BPEL y mejorar sus conjuntos de casos de prueba.

Aprendizaje de nuevas tecnologías

En esta fase, se tomó contacto con las nuevas tecnologías necesarias para la realización de este PFC (Ver 4.4.4), entre las que cabe destacar WS-BPEL, XSLT y XPath.

La mayor parte de estas tecnologías era totalmente desconocida para el alumno, por lo que su aprendizaje fue costoso.

Definición de operadores

En esta fase, se realizó un estudio donde se definieron los operadores para WS-BPEL que forman parte de este PFC: Los operadores de cobertura (CFA, CDE, CCO y CDC) y los operadores que en circunstancias especiales ayudan a aplicar criterios de cobertura (EIU, EIN, EAP y EAN). En este estudio tambien se anañizó la equivalencia de los operadores definidos con los existentes para otros lenguajes estructurados, como C, Fortran o Ada, durante un periodo de investigación que abarca los meses de marzo y abril de 2011 (ver 3.1).

Operador CFA

En esta fase se desarrolló el operador de cobertura de sentencias.

Esta es la fase más compleja de todo el calendario, ya que se manejaron varias opciones de implementación para este operador. En su definición, este operador sustituía una actividad por una excepción (actividad `throw`). Luego, tras varias reuniones, varios miembros del grupo recomendaron que, además de realizar esa sustitución, debería de instalar un manejador de fallos (actividad `catch` dentro de la actividad `faultHandlers`) para recoger la excepción añadida. Finalmente, se decidió emplear la actividad `exit` para terminar abruptamente la composición, opción más simple y que cumple los requisitos del operador. Por este motivo, se realizó en paralelo a las siguientes fases.

Operador EIU

En esta fase se desarrolló el operador EIU.

Dada su relación con los operadores EIN, EAP y EAN, se decidió desarrollar paralelamente estos cuatro operadores.

Operador EIN

En esta fase se desarrolló el operador EIN.

Operador EAP

En esta fase se desarrolló el operador EAP.

Operador EAN

En esta fase se desarrolló el operador EAN.

Operador CDE

En esta fase se desarrolló el operador de cobertura de decisión.

Al igual que ocurrió con los operadores EIU, EIN, EAP y EAN, se decidió desarrollar concurrentemente este operador junto a los operadores CCO y CDC, dada su estrecha relación.

Operador CCO

En esta fase se desarrolló el operador de cobertura de condición.

Operador CDC

En esta fase se desarrolló el operador de cobertura de decisión/condición.

Redacción de la memoria

En esta fase, se redactó esta documentación. Comenzó una vez terminadas las fases anteriores, para así tener conocimiento suficiente para poder escribirla adecuadamente.

Adaptación de la composición *LoanApproval*

En esta fase, se adaptó la composición *LoanApproval* (ver 5.5.4) para que se le pudieran aplicar los operadores desarrollados.

Creación de los casos de pruebas para *LoanApproval* modificada

En esta fase, se creó el conjunto de casos de prueba para la composición *LoanApproval* modificada. Esta fase se realizó a la vez que la anterior, pues la adaptación requería de un conjunto de casos de prueba para comprobar su corrección.

Periodo de investigación

En esta fase se estudió si era necesario definir e implementar un operador para WS-BPEL 2.0 para aplicar el criterio de cobertura múltiple. Este estudio abarcó la segunda mitad del mes de Julio de 2011 (Ver figura 3.1).

Como conclusión, se extrajo que, dadas las propiedades existentes en las expresiones XPath, con el criterio de cobertura de decisión/condición logramos ejercitar todos los valores de la tabla de verdad de la decisión, objetivo del criterio de cobertura múltiple. No obstante, como trabajo futuro queda realizar un estudio más profundo sobre esta cuestión.

Revisión de todos los operadores

En esta fase se comprobó que los operadores desarrollados/implementados en este PFC satisfacen los requisitos pedidos, a partir de las pruebas unitarias definidas previamente.

Esta fase fue la última, con el objeto de realizar la depuración de manera conjunta.

3.2. Gestión del tiempo y recursos

En la figura 3.1 se presenta el diagrama de Gantt que refleja la gestión del tiempo empleado para realizar este PFC. Se pueden distinguir claramente las fases del proyecto, representadas como tareas. Se puede apreciar gráficamente la concurrencia entre algunas fases, por su similitud, como se especificó con anterioridad.

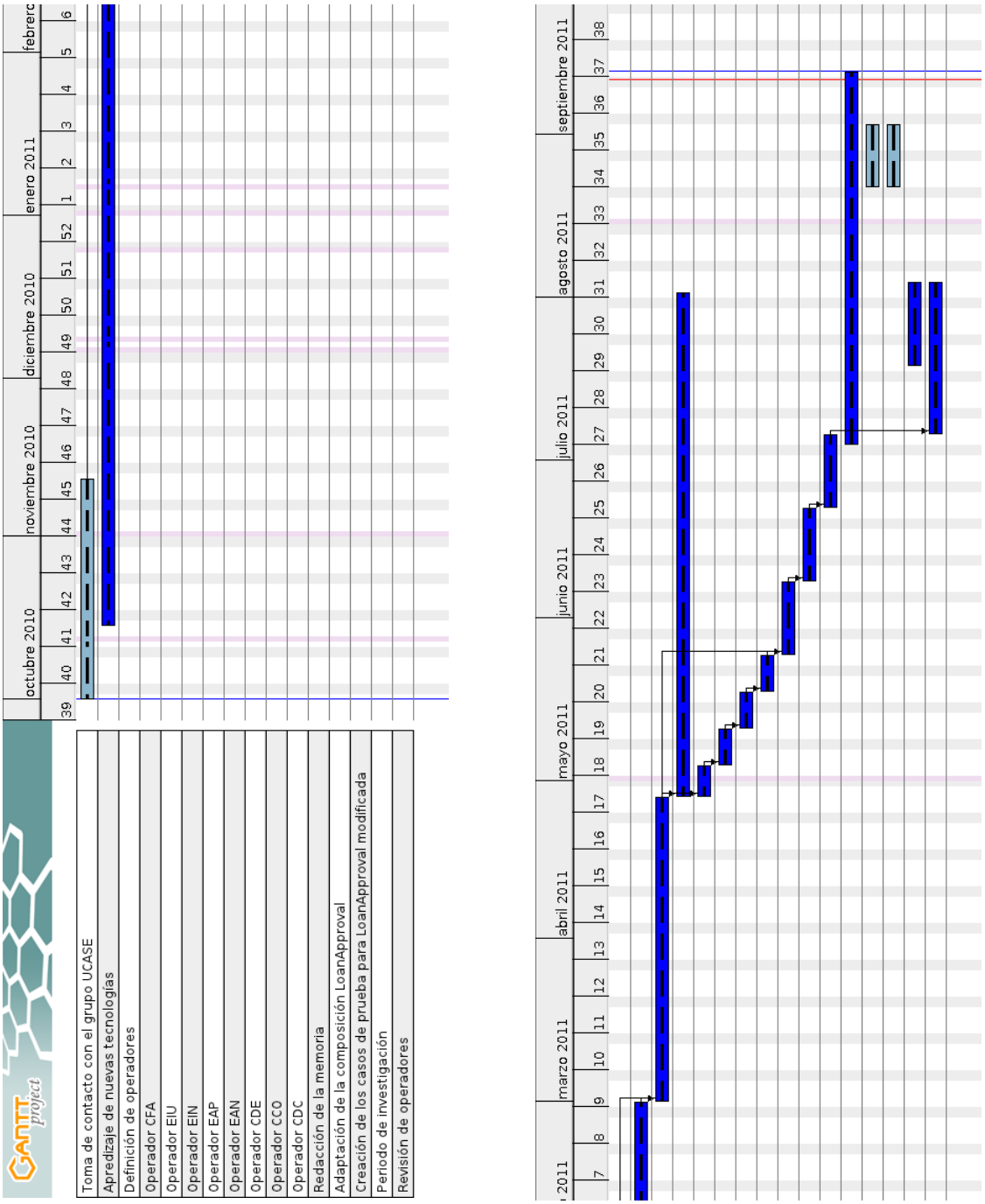

Figura 3.1.: Diagrama de Gantt

4. Descripción general del proyecto

4.1. Perspectiva del producto

4.1.1. Entorno del producto

La herramienta **MuBPEL** [5] forma parte de GAmera [19]. GAmera es una herramienta orientada a la mutación evolutiva, de manera que sólo genera un subconjunto de mutantes, seleccionado mediante un algoritmo genético. Dicha herramienta está formada por:

- Un algoritmo genético que selecciona los mutantes.
- Una herramienta que los compara con la composición original, comprobando si cambia el comportamiento respectivamente. Aquí es donde se encuadra la herramienta **MuBPEL**.

La herramienta **MuBPEL** posee las siguientes dependencias:

- Por parte de GAmera, depende de Java 6 [38], Tomcat 5.5 [23], ActiveBPEL 4.1 [4], XMLBeans 2.1.0 [24], BPELUnit 1.4 [1], Saxon-B 9.1.0.1 [30], galib 2.4.7 [34] y JUnit 4 [39].
- Específicamente, depende de los módulos AnalizadorXPath, ConversorXPath e InstrumentadorBPEL, disponibles en `https://neptuno.uca.es/redmine/projects/sources-fm/repository/show/trunk/src`.

4.1.2. Interfaz de usuario

La herramienta **MuBPEL** no posee GUI. La herramienta se emplea a través de la línea de órdenes:

```
mubpel (argumentos).
```

En el apéndice B están disponibles los comandos de los que dispone **MuBPEL**, además de los argumentos necesarios.

4.2. Funciones

Las funciones de **MuBPEL** son:

- Analizar una composición WS-BPEL, es decir, identificar las actividades que pueden ser mutadas.
- Aplicar un operador a la composición WS-BPEL para generar un mutante.
- Aplicar todos los operadores disponibles a la composición WS-BPEL, generando todos los mutantes posibles.
- Ejecutar una composición WS-BPEL frente al conjunto de casos de prueba, comprobando si se han superado con éxito.
- Ejecutar los mutantes y comparar la salida de estos frente a la del programa original hasta que se encuentre la primera diferencia.
- Ejecutar los mutantes y comparar la salida de estos frente a la del programa original.
- Comparar dos resultados de ejecución de composiciones WS-BPEL.
- Normalizar una composición WS-BPEL.

4.3. Características del usuario

Dado el objetivo de esta herramienta, que es el de generar un conjunto de mutantes de una composición WS-BPEL, comparando la salida de la misma con la de cada uno de ellos, es necesario que el usuario conozca como mínimo:

- El lenguaje WS-BPEL 2.0 [36].
- La prueba de mutaciones [28].
- El análisis de mutaciones. [28]
- Los criterios de cobertura de código. [35]

4.4. Restricciones generales

4.4.1. Control de versiones

Para el desarrollo de este PFC, se ha empleado un sistema de control de versiones, lo cual tiene las siguientes ventajas:

- Sirve para respaldar los datos que se poseen del proyecto.
- Permite revertir cambios perjudiciales en el proyecto, ya que se disponen de todas las versiones del mismo.

Subversion

En concreto, se ha empleado el sistema de control de versiones *subversion*.

Este SCV es un sistema centralizado, ya que los cambios son enviados a un servidor, ampliamente utilizado por desarrolladores.

Las órdenes más útiles a la hora de emplear este SCV son las siguientes:

- Para descargar una copia de trabajo de un repositorio existente, se ejecuta:
  `svn co`
- Para actualizar la copia de trabajo, se ejecuta:
  `svn up`
- Para enviar los cambios realizados localmente, se ejecuta:
  `svn ci –m "Cambios realizados"`
  Mediante la opción -m se indica un mensaje con los cambios que se han realizado en la revisión enviada.

Para obtener más información sobre éstas y otras órdenes disponibles, se puede consultar el manual [15].

Éste es el manejo de *subversion* mediante línea de órdenes. Si se desea un manejo más cómodo, existen varias interfaces gráficas que permiten una mayor comodidad a la hora de interactuar con el mismo. En este caso, se empleó RapidSVN. Podemos ver en la figura 4.1 una instantánea de la misma.

Esta interfaz hace más sencillo el manejo de *subversion*, y permite interactuar con el repositorio de manera más intuitiva, dando la opción de seleccionar los cambios que queramos actualizar de manera rápida y simple. Para obtener la última versión, se puede visitar su sitio web [3].

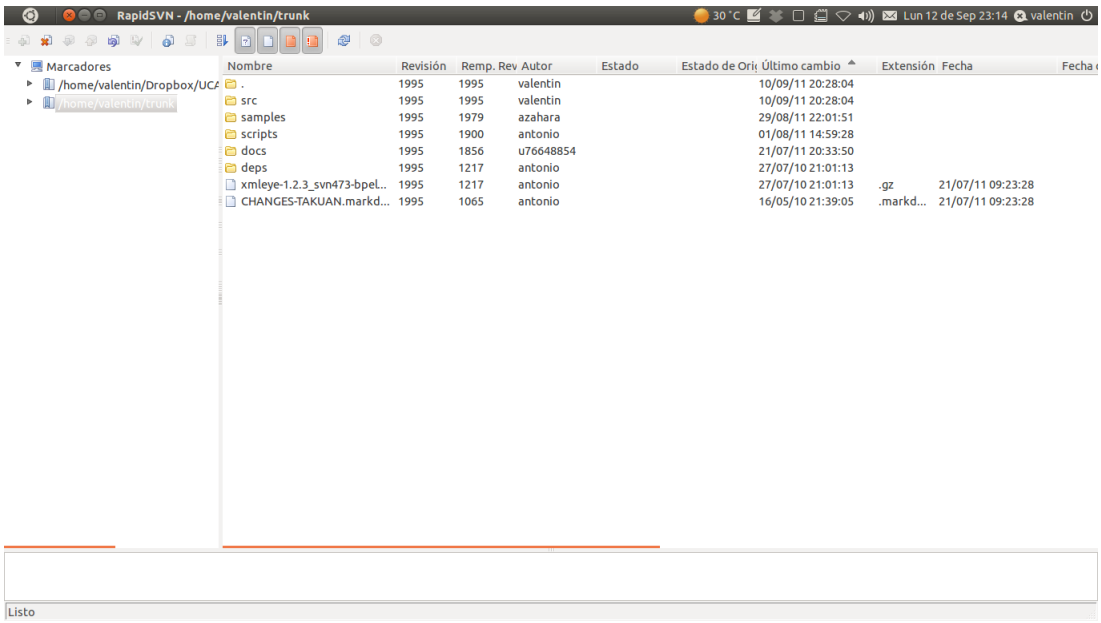

Figura 4.1.: RapidSVN

4.4.2. Servidor de integración continua

La integración continua es una metodología propuesta por Martin Fowler [26] que consiste en integrar los cambios realizados en un proyecto software con el objeto de detectar los fallos lo más pronto posible.

Para llevar a cabo esta metodología, se implanta un servidor de integración continua el cual, cada cierto tiempo, realiza lo siguiente:

1. Obtener, mediante el SCV correspondiente, la última versión del código fuente.
2. Compilar el código obtenido y ejecutar las pruebas unitarias.
3. Generar un informe con los resultados.

Para realizar la compilación y ejecutar las pruebas, el servidor se apoya en herramientas como Apache Maven, las cuales se encargan de la gestión de los proyectos software.

Jenkins

En concreto, se ha empleado el servidor de integración continua Jenkins [14]. En la figura 4.2 podemos ver cómo se organiza dicho servidor.

Este servidor se encarga de realizar las tareas explicadas anteriormente, cada varias horas y cada vez que se registran nuevos cambios en el código. Una vez finaliza su

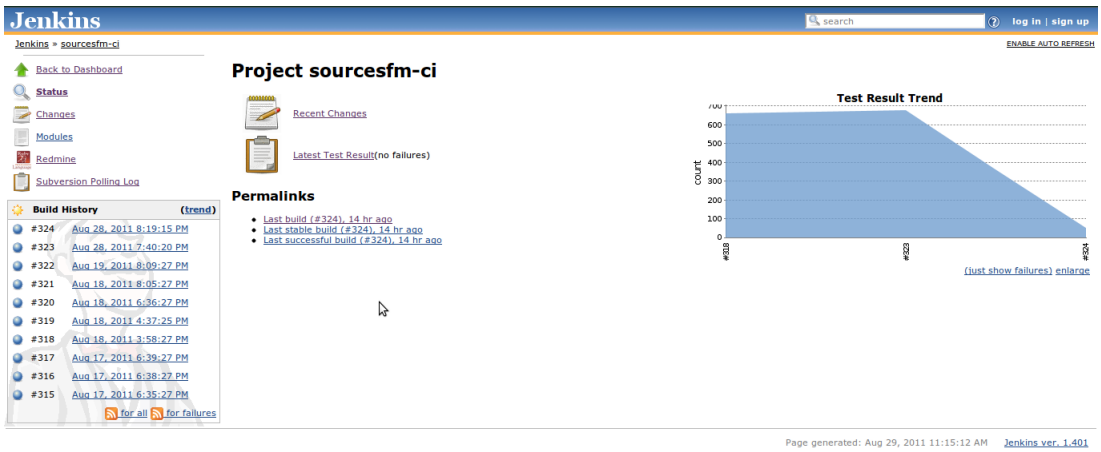

Figura 4.2.: Jenkins

trabajo, en el caso de existir algún error, ya sea de compilación o porque exista alguna prueba que no haya sido superada, envía un correo al desarrollador para avisarle de que esta versión no es estable, indicando el porqué para que éste pueda arreglar los fallos existentes.

4.4.3. Calidad del código fuente

El uso de métricas para el control de la calidad del código fuente es bastante útil a la hora de obtener un código más robusto. Mediante estas métricas, se pueden obtener estadísticas acerca del número de clases, métodos y atributos, permitiendo un empleo óptimo de las mismas.

Sonar

Una herramienta que sirve para la generación de métricas para código fuente Java es Sonar . En la figura 4.3 vemos la estructura principal de esta aplicación.

Esta herramienta muestra los errores que existen dentro del código. Los errores están clasificados como *error* (*blocker*, *critical*), *warning* (*major*, *minor*) e *info*. Están ordenados en función de su importancia, siendo el error *blocker* el más grave, y el error *info* el menos importante.

También se muestran estadísticas acerca de las líneas de código escritas, del número de clases empleadas y de las dependencias que tienen las clases, entre otros.

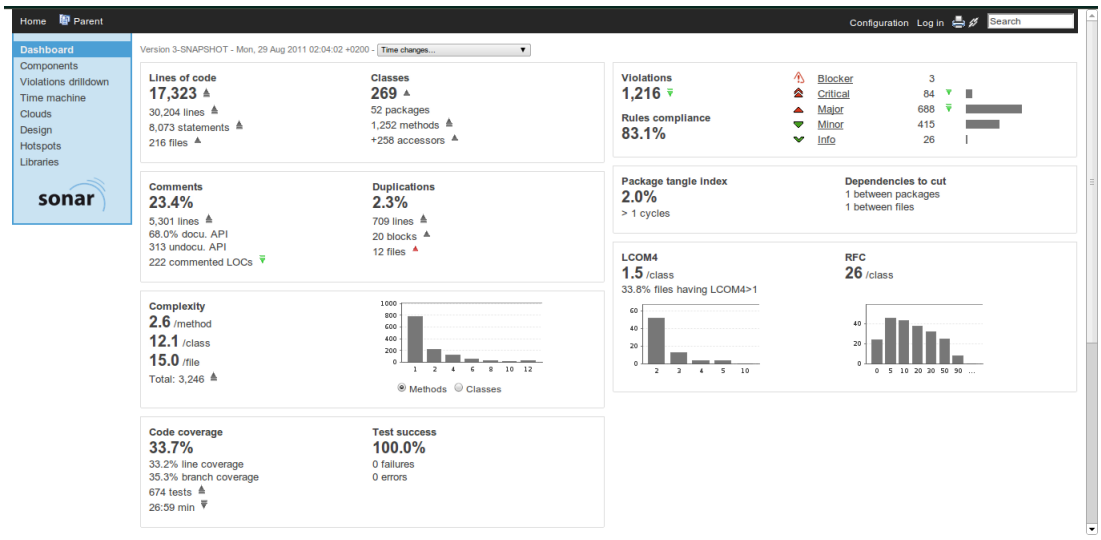

Figura 4.3.: Sonar

4.4.4. Lenguajes de programación y tecnologías

Los lenguajes de programación y las tecnologías que se han utilizado durante el desarrollo de este PFC son:

- WSDL: Se emplea para la descripción de servicios Web.
- SOAP: Se emplea para el intercambio de mensajes de forma distribuida.
- WS-BPEL: Se emplea para la composición de servicios Web.
- XSLT: Se emplea para la transformación de documentos XML en otros, mediante plantillas.
- XPath: Se emplea para el recorrido de documentos XML.
- XML Schema: Se emplea para la descripción de la estructura de documentos XML.
- JUnit: Se emplea para la realización de pruebas unitarias en clases Java.
- BPELUnit: Se emplea para la realización de pruebas unitarias en composiciones WS-BPEL.

4.4.5. Herramientas

Para desarrollar este PFC se han utilizado las siguientes herramientas:

- Apache Maven [22]: Sistema de gestión de proyetos software, encargado de compilar, ejecutar pruebas y obtener la documentación correspondiente.

- Eclipse [25]: Entorno de desarrollo multiplataforma empleado principalmente en el desarrollo de aplicaciones Java, aunque existen multitud de extensiones que le permiten ser útil para el desarrollo bajo cualquier lenguaje de programación.
- Meld [2]: Visor de diferencias gráfico.
- XMLEye [17]: Visor de documentos XML gráfico, que permite navegar con gran comodidad a través de los mismos.
- Xacobeo [6]: Aplicación que permite ejecutar consultas XPath de manera gráfica, muy útil para comprobar la corrección y validez de las mismas.

4.4.6. Sistemas operativos y hardware

Este PFC ha sido desarrollado y probado en GNU-Linux, empleándose en concreto la distribución Ubuntu, en su versión 11.04.

En cuanto a los requisitos hardware, se recomienda ejecutar la herramienta MuBPEL en un ordenador que disponga de una memoria RAM de, al menos, 1 GiB de capacidad y de un procesador con más de un núcleo, ya que consume bastantes recursos y de esta forma se puede aprovechar el paralelismo a la hora de ejecutar mutantes, levantando varias instancias de ActiveBPEL [4].

5. Desarrollo del proyecto

En este capítulo se recoge el proceso de desarrollo, es decir, las fases de análisis, diseño, implementación y prueba de este PFC . Cabe destacar que las fases de análisis y diseño recogen la última iteración del producto, únicamente, al ser la que contiene la arquitectura del programa al completo.

5.1. Modelo de ciclo de vida

El modelo de ciclo de vida elegido para este PFCes el modelo de ciclo de vida incremental. En la figura 5.1 podemos ver las principales características de este modelo. Está basado en el modelo lineal-secuencial, añadiéndole la capacidad de evolucionar, ya que las fases de análisis, diseño, implementación y prueba no sólo ocurren una vez.

Se parte de unos requisitos iniciales, y tras cada iteración, se obtiene un producto software que puede ser utilizado. No es necesario esperar al final del proceso para obtener un producto software, al contrario que en el modelo lineal-secuencial. Además, no es necesario que los requisitos estén completamente definidos al comienzo, pues pueden redefinirse tras cada iteración, permitiendo añadir nuevas funcionalidades al software.

Este modelo se adapta perfectamente a nuestras necesidades, ya que en cada iteración, partimos de un conjunto de operadores ya definido. Tras las fases de análisis, diseño, implementación y prueba, obtendremos una nueva versión del conjunto de operadores, al cual se le ha añadido un operador nuevo. De esta forma, podemos utilizar **MuBPEL** sin tener que esperar a que estén terminados todos los operadores.

5.2. Herramienta de modelado empleada: *dia*

La herramienta *dia* nos permite dibujar los diferentes tipos de diagramas UML existentes, como los diagramas de casos de uso, de clases o de secuencia. Además, permite dibujar otro tipo de diagramas, como los de entidad-relación o de flujo.

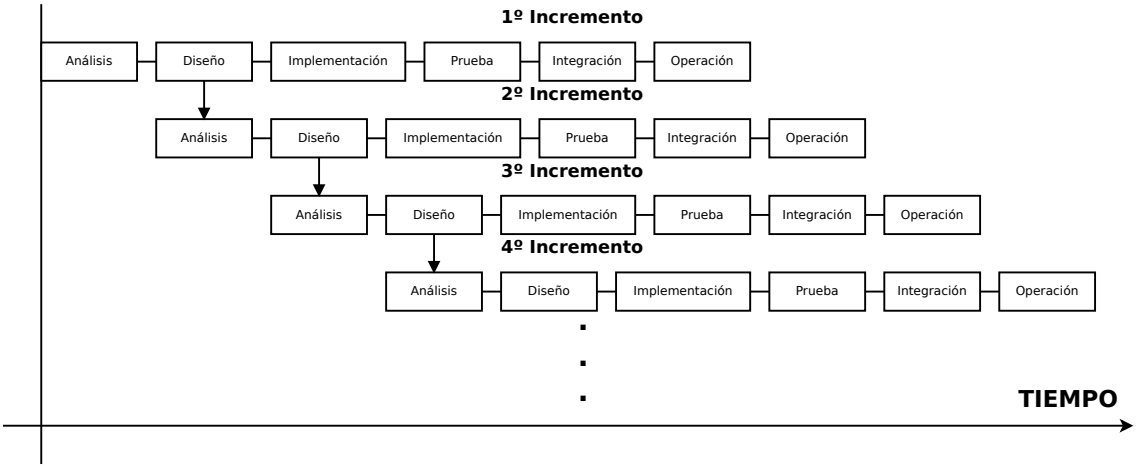

Figura 5.1.: Modelo de ciclo de vida incremental

Tiene la ventaja de ser personalizable, pues mediante un archivo XML se pueden añadir nuevas formas que podemos emplear en nuestros diagramas, de una manera análoga a un archivo *svg*.

Los diagramas generados con esta herramienta pueden ser exportados a una gran cantidad de formatos, como por ejemplo *png*, *svg*, *eps*, o directamente a código LaTeX.

Además, se puede generar el código Java, C++, Pascal y Python correspondiente a los diagramas generados, exportándolos directamente.

La herramienta *dia* está disponible para Windows, Linux y Mac OS X, y se puede descargar en: http://dia-installer.de/.

5.3. Requisitos

5.3.1. Funcionales

- Analizar una composición WS-BPEL. Esto consiste en buscar las instrucciones del programa que pueden ser mutadas. Como resultado, se debe generar un fichero en el que se indique, por cada operador, el número de instrucciones donde puede ser aplicado y el valor máximo del atributo, es decir, el máximo número de tipos de cambios que puede hacerse.
- Generar un mutante aplicando un operador a la composición original. Se debe indicar para ello el operador, la instrucción que se desea mutar y, si procede, el valor del atributo. Como resultado, se debe generar un mutante, es decir, un fichero con el mismo contenido que el programa original a excepción de la instrucción mutada.

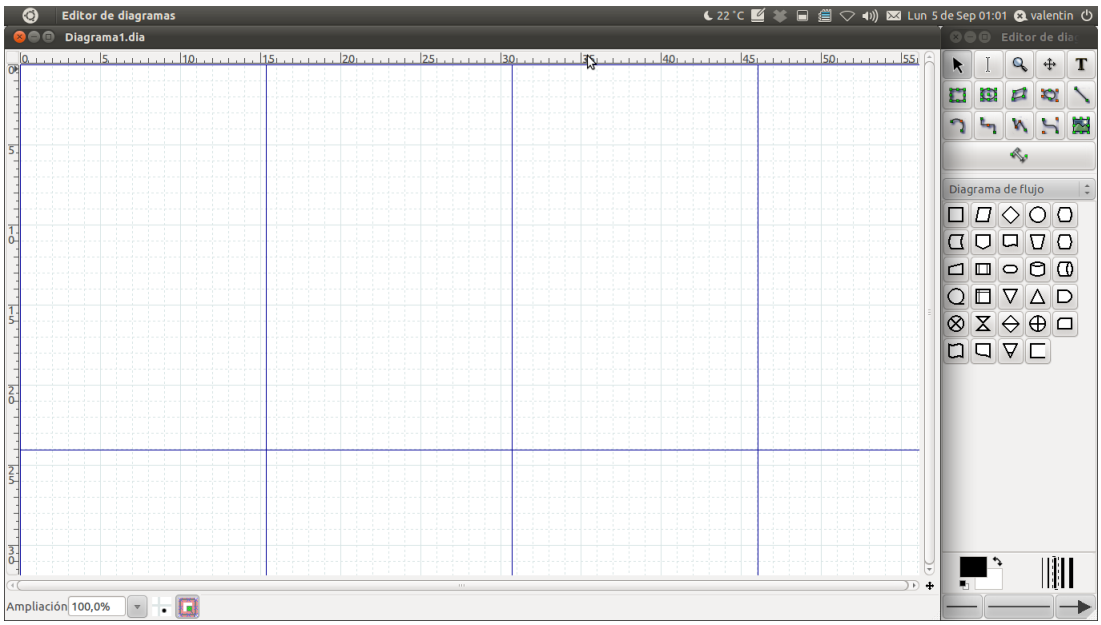

Figura 5.2.: Herramienta de modelo *dia*

- Generar todos los mutantes aplicando todos los operadores. Como resultado, se deben generar tantos mutantes como sea posible. Esto viene determinado por el número de operadores, el número de instrucciones que pueden ser cambiadas y por el tipo de cambio que se pueda hacer en cada caso.

- Ejecutar una composición WS-BPEL frente a su conjunto de casos de prueba. Como resultado, se debe generar un fichero con los resultados de la ejecución, que indicarán si los casos de prueba se han pasado con éxito o no.

- Ejecutar los mutantes y comparar su salida con la de la composición original, hasta encontrar la primera diferencia. Como resultado, se obtiene un fichero en el que, para cada mutante, se indica el resultado de la comparación.

- Ejecutar los mutantes y comparar su salida con la de la composición original. Como resultado, se obtiene un fichero en el que, para cada mutante, se indica el resultado de la comparación.

- Comparar dos salidas de ejecución de una composición WS-BPEL, sean de la composición original o de un mutante. Como resultado, se debe obtener un fichero en el que se indican las diferencias entre ambas.

- Normalizar una composición WS-BPEL. Como resultado, se obtiene un fichero con la composición WS-BPEL normalizada, en su forma canónica.

5.3.2. De información

- El sistema debe guardar el nombre de cada operador de mutación, así como el valor numérico del mismo, perteneciente al intervalo $[1, 34]$, debido que existe un total de 34 operadores de mutación. También se debe almacenar el valor máximo del atributo en cada caso.
- Los individuos se representan por el operador, el operando y el atributo.
- La composición original, el conjunto de casos de prueba y las salidas de ejecución se deben almacenar en ficheros, guardando su nombre y ruta.

5.3.3. De reglas de negocio

Las mutaciones realizadas serán de orden 1, es decir, un mutante se generará tras aplicar un único operador en una única instrucción y con un valor de atributo (en el caso en que proceda), a la composición original.

5.3.4. De interfaz

La herramienta **MuBPEL** forma parte de la herramienta GAmera [19], ya que los operadores de mutación son fundamentales en la generación de mutantes, los cuales se encarga de seleccionar GAmera mediante su algoritmo genético.
Así pues, los resultados generados por **MuBPEL** deben ser completamente compatibles con GAmera, para que de esta forma puedan ser utilizados por la misma.

5.3.5. No funcionales

- Rendimiento. Se busca un rendimiento alto, especialmente a la hora de generar mutantes y realizar las comparación de las salidas de ejecución.
- Fiabilidad. Las mutaciones realizadas han de ser correctas, al igual que la comparación entre las salidas de ejecución, ya que el trabajo posterior se basa principalmente en ambas.
- Mantenibilidad. Se busca que la herramienta sea fácil de mantener y de ampliar, con vistas de añadir nuevos operadores de mutación en un futuro.

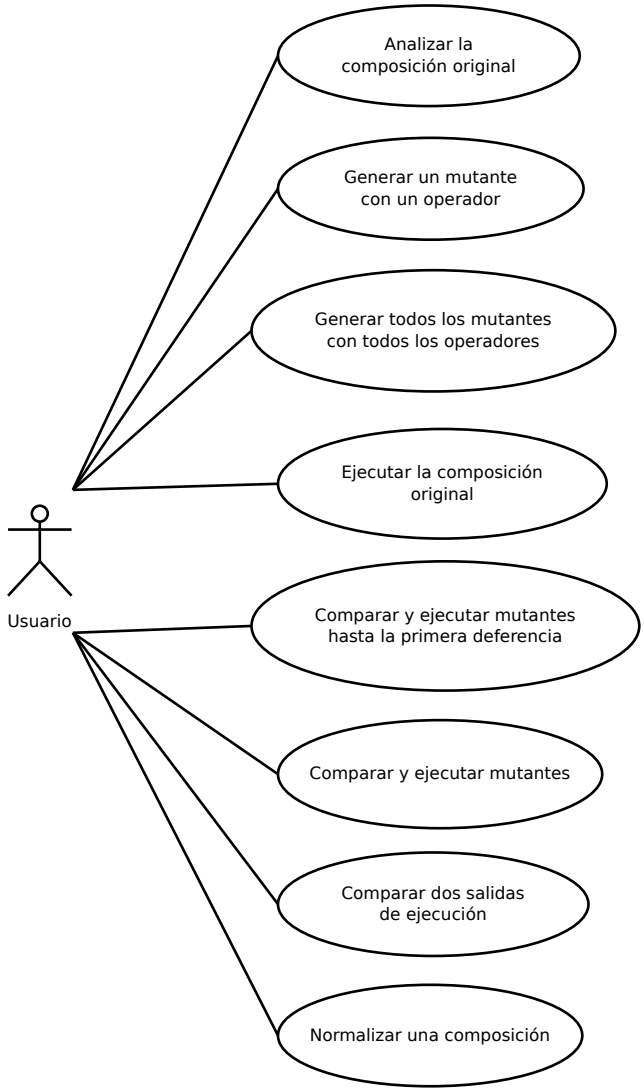

Figura 5.3.: Diagrama de casos de uso

5.4. Análisis

5.4.1. Modelo de casos de uso

Mediante el modelo de casos de uso, podemos reflejar cómo el usuario interactúa con el sistema, además de las funcionalidades del mismo.

En la figura 5.3 podemos ver el diagrama de casos de uso de **MuBPEL**.

A continuación, se facilitan las especificaciones de cada uno de los casos de uso del sistema.

5.4.1.1. Analizar la composición original

Actor principal El usuario, quien desea saber qué operadores son aplicables a la composición, además del número de operandos y valor máximo del atributo para cada uno de ellos.

Precondiciones Es necesario que exista una composición WS-BPEL.

Postcondiciones Se obtiene una lista en la que se muestra cada operador, junto al número de operandos y el valor máximo del atributo.

Escenario principal

1. El usuario selecciona la opción de analizar la composición, especificando la misma.
2. El sistema le muestra una lista con los operadores disponibles, en la que se indica tanto el número de operandos como el valor máximo del atributo para cada uno de ellos.

5.4.1.2. Generar un mutante con un operador

Actor principal El usuario, quien desea obtener un mutante a partir de la composición original, aplicando un operador existente.

Precondiciones Es necesario que exista una composición WS-BPEL.

Postcondiciones Se obtiene el mutante deseado.

Escenario principal

1. El usuario selecciona la opción de generar un mutante en concreto, especificando el operador, el operando y el valor del atributo.
2. El sistema genera el mutante deseado.

Extensiones

2a. El operador no puede ser aplicado a la composición dada.

1. El sistema informa al usuario de que el operador dado no es aplicable a la composición.

5.4.1.3. Generar todos los mutantes con todos los operadores

Actor principal El usuario, quien desea obtener todos los mutantes posibles tras aplicar el conjunto de operadores al completo.

Precondiciones Es necesario que exista una composición WS-BPEL.

Postcondiciones Se obtienen todos los mutantes generados aplicando todos los operadores a la composición original.

Escenario principal

1. El usuario selecciona la opción de generar todos los mutantes, especificando la composición original.
2. El sistema genera el conjunto completo de mutantes, aplicando todos los operadores en todos los operandos, y en el rango de valores que el valor del atributo indique.

Extensiones

2a. Ningún operador puede ser aplicado a la composición dada.

1. El sistema informa al usuario de que ninguno de los operadores son aplicables a la composición.

5.4.1.4. Ejecutar la composición original

Actor principal El usuario, quien desea ejecutar la composición WS-BPEL original frente al conjunto de casos de prueba de la misma.

Precondiciones Es necesario que exista una composición WS-BPEL y un conjunto de casos de prueba de la misma.

Postcondiciones Se obtienen los resultados de cada uno de los casos de prueba, indicando si se pasaron o no con éxito.

Escenario principal

1. El usuario selecciona la opción de ejecutar la composición, indicando la misma y su conjunto de casos de prueba asociado.
2. El sistema muestra los resultados de la ejecución, indicando si hubo o no éxito a la hora de probar la composición.

5.4.1.5. Comparar y ejecutar mutantes hasta la primera diferencia

Actor principal El usuario, quien desea comparar la salida de ejecución de la composición original con la de los mutantes, hasta encontrar la primera diferencia entre ellas.

Precondiciones Es necesario que exista una composición WS-BPEL, un conjunto de casos de prueba de la misma y, al menos, un mutante de dicha composición.

Postcondiciones Se obtiene para cada mutante el resultado de la comparación, es decir, si está vivo, muerto o es erróneo.

Escenario principal

1. El usuario selecciona la opción de comparar hasta la primera diferencia, indicando la composición original, el conjunto de casos de prueba, la salida de ejecución de la composición original y los mutantes.
2. El sistema compara la salida de la composición original con la de cada mutante, deteniendo el proceso al encontrar la primera diferencia.

5.4.1.6. Comparar y ejecutar mutantes

Actor principal El usuario, quien desea comparar la salida de ejecución de la composición original con la de cada uno de los mutantes.

Precondiciones Es necesario que exista una composición WS-BPEL, un conjunto de casos de prueba de la misma y, al menos, un mutante de dicha composición.

Postcondiciones Se obtiene para cada mutante el resultado de la comparación, es decir, si está vivo, muerto o es erróneo.

Escenario principal

1. El usuario selecciona la opción de comparar, indicando la composición original, el conjunto de casos de prueba, la salida de ejecución de la composición original y los mutantes.
2. El sistema compara la salida de la composición original con la de cada uno de los mutantes al completo.

5.4.1.7. Comparar dos salidas de ejecución

Actor principal El usuario, quien desea comparar dos salidas de ejecución de una composición WS-BPEL.

Precondiciones Es necesario que existan dos salidas de ejecución.

Postcondiciones Se obtienen los resultados de la comparación de ambas salidas, indicando las diferencias en el caso oportuno.

Escenario principal

1. El usuario selecciona la opción de comparar dos salidas de ejecución, facilitando ambas.
2. El sistema compara ambas salidas, mostrando al usuario los resultados.

Extensiones

2a. El número de casos de prueba de ambas salidas difiere.

1. El sistema informa al usuario de que la comparación no puede realizarse al no tener ambas salidas el mismo número de casos de prueba.

5.4.1.8. Normalizar una composición

Actor principal El usuario, quien desea obtener la forma canónica de una composición WS-BPEL.

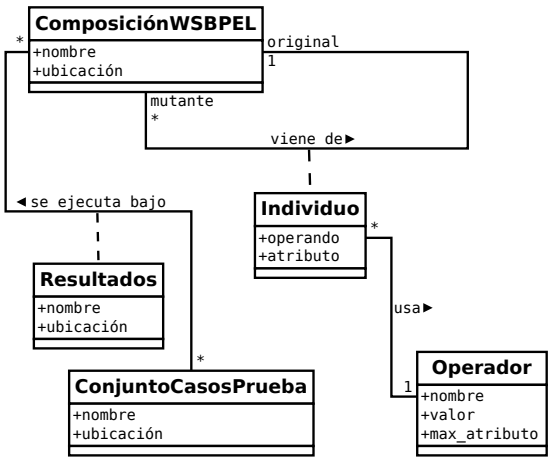

Figura 5.4.: Diagrama de clases conceptuales

Precondiciones Es necesario que exista una composición WS-BPEL.

Postcondiciones Se obtiene la composición WS-BPEL normalizada.

Escenario principal

1. El usuario selecciona la opción de normalizar, indicando la composición que desea normalizar.
2. El sistema genera la forma canónica de la composición dada.

5.4.2. Modelo conceptual de datos

Mediante el modelo conceptual de datos, obtenemos cuáles son las clases que componen nuestro sistema, así como las relaciones entre las mismas.

En la figura 5.4 podemos ver el diagrama de clases conceptuales correspondiente a la herramienta **MuBPEL**.

Una *ComposiciónWS-BPEL* posee un nombre y una ubicación en disco. Posee los roles tanto de composición original como de mutante, ya que esta clase representa a ambas.

Un mutante se relaciona con la composición original a través de un *Individuo*, el cual posee el operando y el valor del atributo utilizado en la mutación. A su vez, el *Individuo* genera el mutante mediante un *Operador*, el cual posee un nombre de tres letras, un valor comprendido entre 1 y 34, el número de operadores existentes, y el valor máximo del atributo, que dependerá del operador en cuestión.

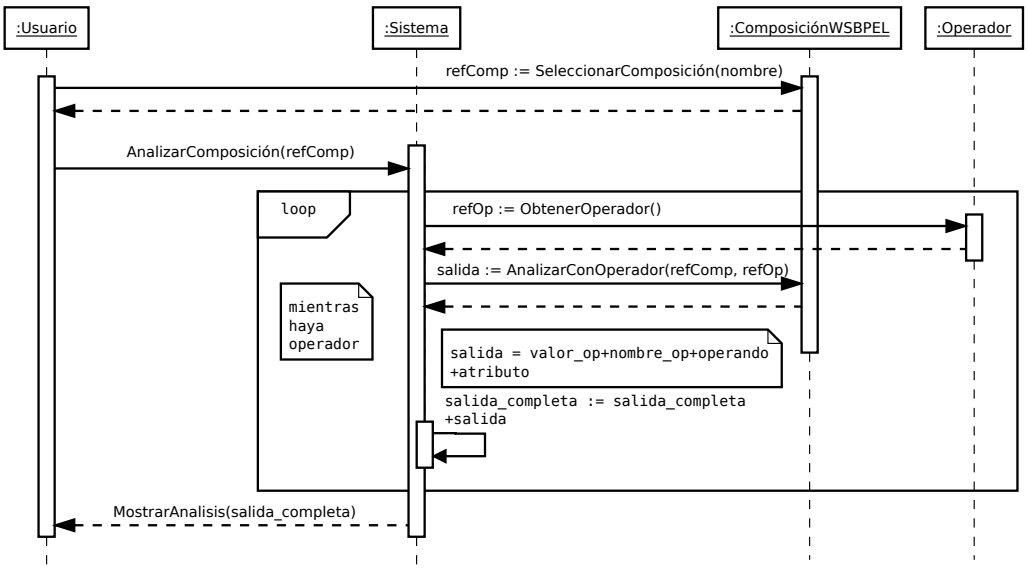

Figura 5.5.: Diagrama de secuencia de Analizar Composición

La composición original se ejecuta bajo un *ConjuntoCasosPrueba*, el cual posee un nombre y una ubicación en disco. Tras realizar la ejecución, se obtienen unos *Resultados*, los cuales pertenecen a la relación entre ComposiciónWSBPEL y *ConjuntoCasosPrueba*. Estos *Resultados* tendrán un nombre y una ubicación, de igual manera.

5.4.3. Diagramas de secuencia

A continuación, se presentan los diagramas de secuencia correspondientes a cada caso de uso detallado anteriormente. Mediante estos diagramas, podemos identificar de manera intuitiva las operaciones que tendrá nuestro sistema. Así pues, una vez identificadas, se añaden los contratos de las mismas.

5.4.3.1. Analizar composición

Operación refComp := SeleccionarComposición(nombre)

Responsabilidades Selecciona una composición WS-BPEL existente.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existe una composición WS-BPEL cuyo nombre es igual a *nombre*.

Postcondiciones Se obtiene una referencia *refComp* a un objeto ComposiciónWSBPEL, que representa a la composición seleccionada.

Operación AnalizarComposición(refComp)

Responsabilidades Analiza una composición WS-BPEL.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existe una referencia a objeto ComposiciónWSBPEL *refComp*.

Postcondiciones Se obtiene el análisis de la composición dada.

Operación refOp := ObtenerOperador()

Responsabilidades Obtiene un operador de mutación dentro del conjunto de los operadores disponibles.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existe al menos un operador de mutación disponible.

Postcondiciones Se obtiene una referencia *refOp* a un objeto de la clase Operador.

Operación salida := AnalizarConOperador(refComp, refOp)

Responsabilidades Analiza los operandos donde puede ser aplicado un operador en una composición dada.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existe una referencia *refComp* a un objeto ComposiciónWSBPEL válida y una referencia *refOp* a un objeto Operador válida.

Postcondiciones Se obtiene el número de operandos y valor máximo del atributo para el operador dado en dicha composición.

Operación MostrarAnalisisSalida(salida_completa)

Responsabilidades Muestra la salida obtenida tras el análisis.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Ninguna.

Postcondiciones Muestra la salida obtenida tras analizar la composición, es decir, los operadores, junto al número de operandos y valor máximo del atributo.

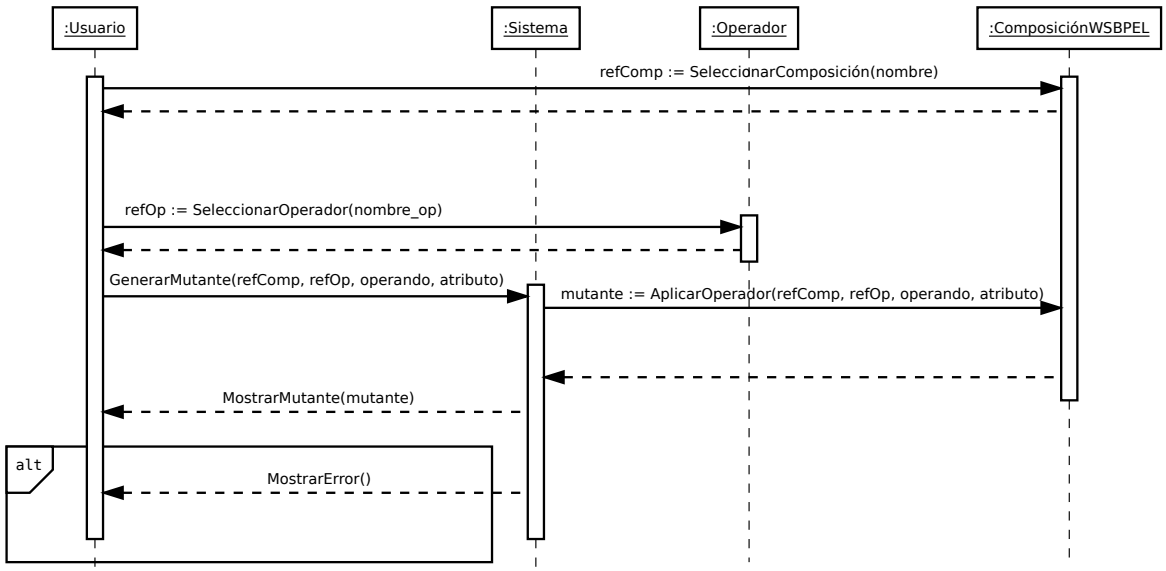

Figura 5.6.: Diagrama de secuencia de Generar un mutante con un operador

5.4.3.2. Generar un mutante con un operador

Operación refOp := SeleccionarOperador(nombre_op)

Responsabilidades Selecciona un operador de mutación existente.

Referencias cruzadas Véase 5.4.1.2.

Precondiciones Existe un operador de mutación entre los disponibles cuyo nombre es igual a *nombre*.

Postcondiciones Se obtiene una referencia *refOp* a un objeto Operador.

Operación GenerarMutante(refComp, refOp, operando, atributo)

Responsabilidades Genera un mutante a partir de la composición original.

Referencias cruzadas Véase 5.4.1.2.

Precondiciones Existen referencias refComp y refOp a objetos ComposiciónWSBPEL y Operador, respectivamente, válidas, y el operando y valor del atributo.

Postcondiciones Se obtiene un mutante de la composición original.

Operación mutante := AplicarOperador(refComp, refOp, operando, atributo)

Responsabilidades Aplica un operador a la composición original.

Referencias cruzadas Véase 5.4.1.2.

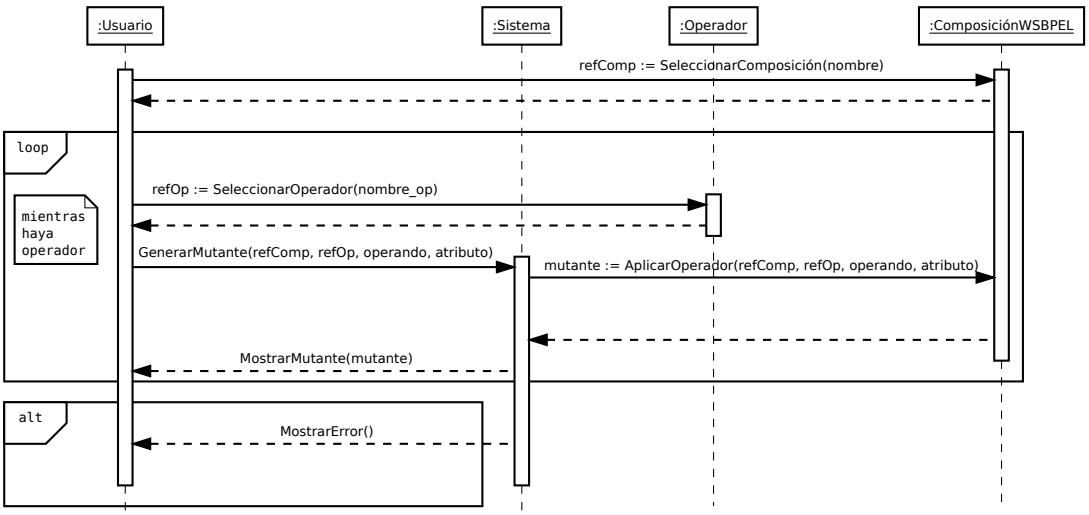

Figura 5.7.: Diagrama de secuencia de Generar todos los mutantes con todos los operadores

Precondiciones Existen referencias refComp y refOp a objetos ComposiciónWSBPEL y Operador, respectivamente, válidas, y el operando y valor del atributo.

Postcondiciones Se le aplica el operador referenciado por refOp a la composición referenciada por refComp, en el operando dado y con el valor de atributo dado, si eran válidos. Si no, se genera un mensaje de error.

Operación MostrarMutante(mutante)

Responsabilidades Muestra la salida obtenida tras el análisis.

Referencias cruzadas Véase 5.4.1.2.

Precondiciones Ninguna.

Postcondiciones Muestra el mutante generado en la salida.

Operación MostrarError()

Responsabilidades Muestra los errores ocurridos durante alguna operación.

Referencias cruzadas Véase 5.4.1.2.

Precondiciones Ninguna.

Postcondiciones Muestra el error generado en la salida.

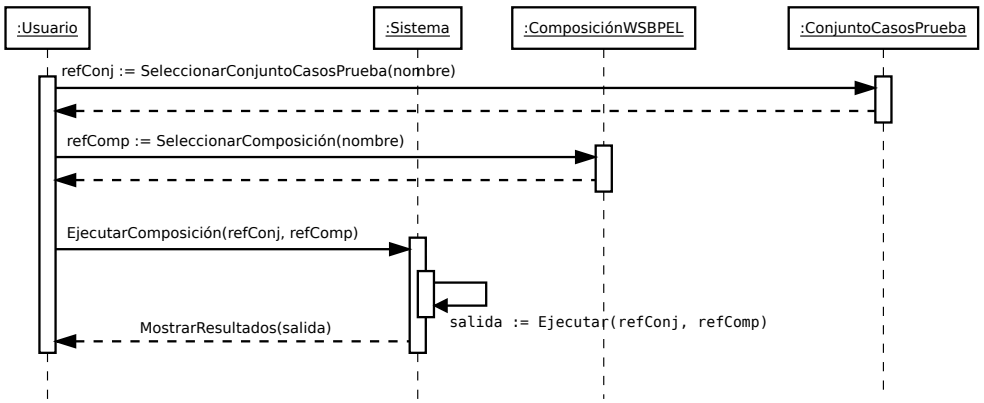

Figura 5.8.: Diagrama de secuencia de Ejecutar la composición original

5.4.3.3. Generar todos los mutantes con todos los operadores

Los contratos de todas las operaciones están presentes en 5.4.3.2.

5.4.3.4. Ejecutar la composición original

Operación refConj := SeleccionarConjuntoCasosPrueba(nombre)

Responsabilidades Selecciona un conjunto de casos de prueba existente.

Referencias cruzadas Véase 5.4.1.4.

Precondiciones Existe un conjunto de casos de prueba cuyo nombre es igual a *nombre*.

Postcondiciones Se obtiene una referencia *refConj* a un objeto ConjuntoCasosPrueba.

Operación EjecutarComposición(refConj, refComp)

Responsabilidades Ejecuta la composición original frente a su conjunto de casos de prueba.

Referencias cruzadas Véase 5.4.1.4.

Precondiciones Existen referencias refConj y refComp a objetos ConjuntoCasosPrueba y ComposiciónWSBPEL, respectivamente, válidas.

Postcondiciones Se obtienen los resultados de la ejecución.

Operación MostrarResultados(salida)

Responsabilidades Muestra la salida obtenida.

Referencias cruzadas Véase 5.4.1.4.

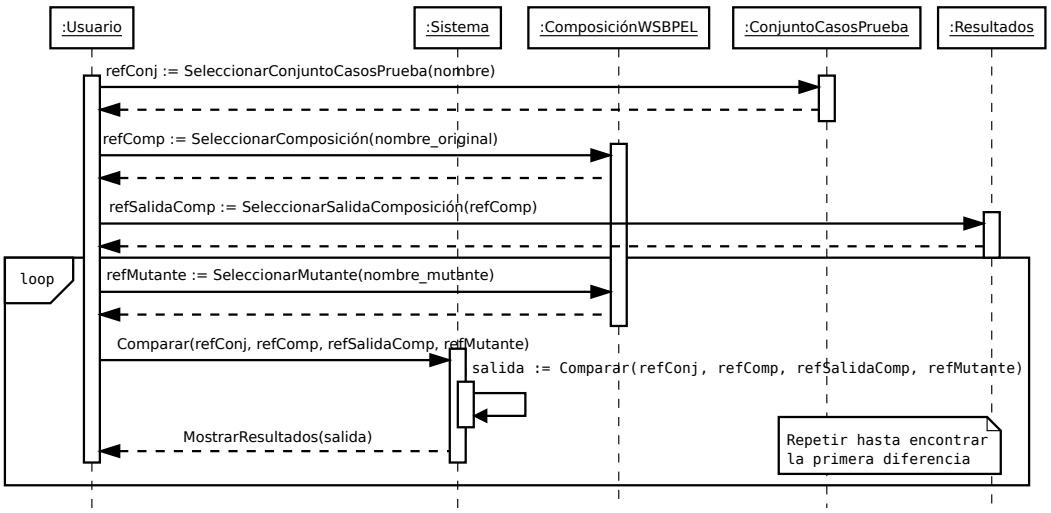

Figura 5.9.: Diagrama de secuencia de Comparar y ejecutar mutantes hasta la primera diferencia

Precondiciones Ninguna.

Postcondiciones Muestra los resultados en la salida.

5.4.3.5. Comparar y ejecutar mutantes hasta la primera diferencia

Operación refSalidaComp := SeleccionarSalidaComposicion(refComp)

Responsabilidades Selecciona un conjunto de casos de prueba existente.

Referencias cruzadas Véase 5.4.1.5.

Precondiciones Existe un conjunto de casos de prueba cuyo nombre es igual a *nombre*.

Postcondiciones Se obtiene una referencia *refConj* a un objeto ConjuntoCasosPrueba.

Operación refMutante := SeleccionarMutante(nombre_mutante)

Responsabilidades Selecciona un mutante existente.

Referencias cruzadas Véase 5.4.1.5.

Precondiciones Existe un mutante cuyo nombre es igual a *nombre_mutante*.

Postcondiciones Se obtiene una referencia *refMutante* a un objeto ComposicionWSB-PEL.

Operación Comparar(refConj, refComp, refSalidaComp, refMutante)

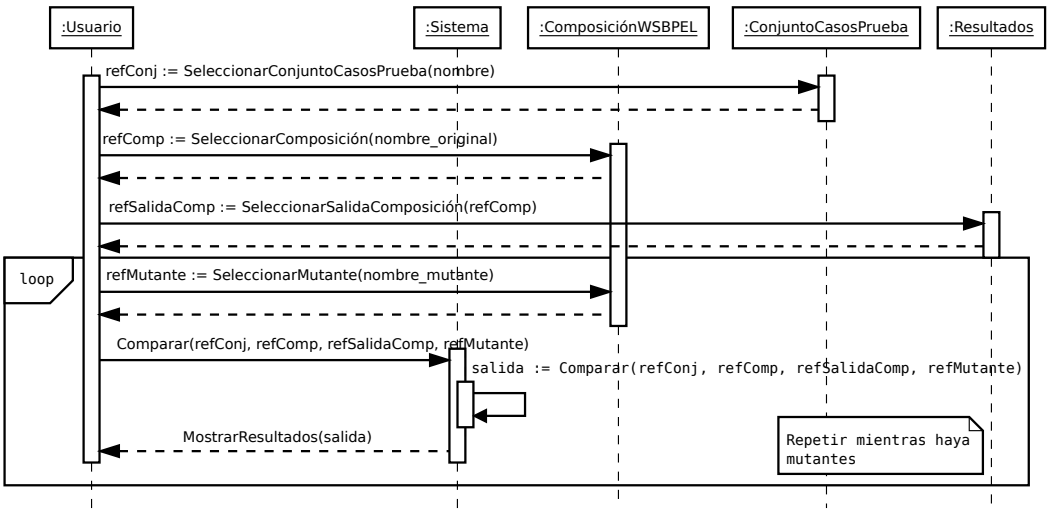

Figura 5.10.: Diagrama de secuencia de Comparar y ejecutar mutantes

Responsabilidades Compara la salida de ejecución de la composición original con la del mutante.

Referencias cruzadas Véase 5.4.1.5.

Precondiciones Existen referencias válidas refConj, refComp, refSalidaComp y refMutante a objetos ConjuntoCasosPrueba, ComposiciónWSBPEL, Resultados y ComposiciónWSBPEL, respectivamente.

Postcondiciones Se obtienen los resultados de la comparación.

5.4.3.6. Comparar y ejecutar mutantes

Los contratos de todas las operaciones están presentes en 5.4.3.5.

5.4.3.7. Comparar dos salidas de ejecución

Operación refSalida := SeleccionarSalidaEjecución(nombre)

Responsabilidades Selecciona la salida de ejecución de una composición.

Referencias cruzadas Véase 5.4.1.7.

Precondiciones Existe una salida de ejecución cuyo nombre es igual a *nombr*e.

Postcondiciones Se obtiene una referencia *refSalida* a un objeto Resultados.

Operación CompararSalidas(refSalida1, refSalida2)

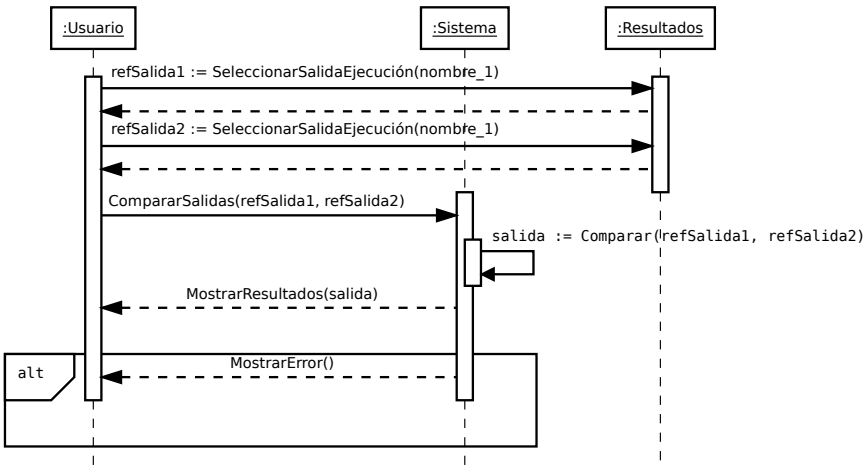

Figura 5.11.: Diagrama de secuencia de Comparar dos salidas de ejecución

Responsabilidades Compara dos salidas de ejecución de una composición.

Referencias cruzadas Véase 5.4.1.7.

Precondiciones Existen referencias refSalida1 y refSalida2 a objetos Resultados válidas.

Postcondiciones Se obtienen los resultados de la comparación. Si no tienen ambas salidas el mismo número de casos de prueba, se genera un mensaje de error.

Operación salida := Comparar(refSalida1, refSalida2)

Responsabilidades Compara dos salidas de un caso de prueba de las salidas de ejecución de una composición.

Referencias cruzadas Véase 5.4.1.7.

Precondiciones Existen referencias refSalida1 y refSalida2 a objetos Resultados válidas.

Postcondiciones Se obtiene el resultado de la comparación.

5.4.3.8. Normalizar una composición

Operación NormalizarComposición(refComp)

Responsabilidades Normaliza una composición WS-BPEL.

Referencias cruzadas Véase 5.4.1.8.

Precondiciones Existe una referencias refComp a un objeto ComposiciónWSBPEL válida.

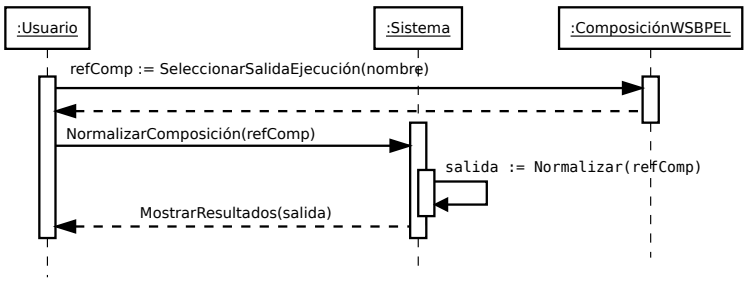

Figura 5.12.: Diagrama de secuencia de Normalizar una composición

Postcondiciones Se obtiene la forma canónica de la composición dada.

Operación salida := Normalizar(refComp)

Responsabilidades Normaliza una composición dada.

Referencias cruzadas Véase 5.4.1.7.

Precondiciones Existe una referencias refComp a un objeto ComposiciónWSBPEL válida.

Postcondiciones Se obtiene el resultado de la normalización.

5.5. Diseño

En esta sección se enmarca este PFC dentro de la herramienta GAmera [19]. Además se resume la utilidad y el objetivo de esta herramienta, para comprender mejor la importancia de la misma.

5.5.1. GAmera

La herramienta GAmera es una herramienta de generación y ejecución de mutantes de composiciones de servicios web para el lenguaje WS-BPEL. Está orientada a la mutación evolutiva, pues tiene la capacidad de seleccionar un subconjunto de mutantes mediante un algoritmo genético. El algoritmo genético selecciona los mutantes de mayor calidad, para reducir el coste computacional que supone ejecutar el conjunto completo de mutantes. La finalidad de esta herramienta es la de obtener un conjunto de casos de prueba de mayor calidad.

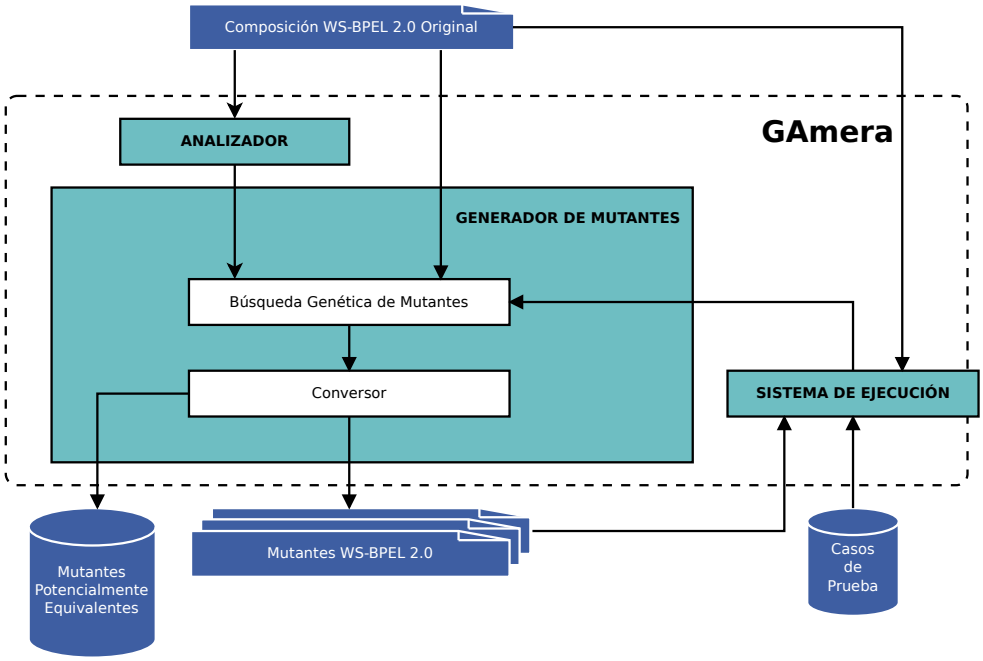

Figura 5.13.: Estructura de GAmera

Estructura

En la figura 5.13 podemos ver la estructura de la herramienta GAmera. Está compuesta por:

Analizador A partir de la composición WS-BPEL original, el analizador se encarga de seleccionar los operadores que pueden ser aplicados a la misma, obteniendo los individuos.

Generador de mutantes Una vez se tiene los operadores aplicables a la composición original, podemos generar todos los mutantes posibles, o un subconjunto de los mismos. Si generamos un subconjunto de ellos, el generador de mutantes seleccionará los individuos mediante un algoritmo genético, cuyo criterio se basa en la existencia de casos de prueba que maten a los mutantes [20]. Una vez seleccionados, el conversor generará a partir de estos individuos los mutantes WS-BPEL, empleando hojas de estilo XSLT para realizar las transformaciones pertinentes.

Ejecución de mutantes Una vez generados, los mutantes pasan al sistema de ejecución, donde se ejecutan frente al conjunto de casos de prueba de la composición. El motor WS-BPEL empleado es ActiveBPEL 4.1 [4]. Se emplea también BPELUnit [1], un framework de pruebas unitarias para WS-BPEL, para describir los casos de prueba. Una vez ejecutados, los mutantes se clasifican en:

Muerto La salida del mutante difiere con la salida de la composición original.

Vivo La salida del mutante coincide con la salida de la composición original.

Erróneo El mutante generado no puede desplegarse correctamente, bien debido a un fallo de implementación del operador de mutación aplicado, o bien porque la mutación aplicada hace que ocurra esto.

5.5.2. Enmarcación de este PFC dentro de GAmera

Este PFC forma parte del generador de mutantes de GAmera, es decir, forma parte de la herramienta **MuBPEL**, ya que para poder realizar la generación de mutantes, es necesaria la presencia de los operadores de mutación, pues son los encargados de ello. A través de este proyecto, se introduce una nueva categoría de operadores de mutación, los operadores de cobertura (ver 1.3). Concretamente, se implementan los siguientes:

CFA Aplica el criterio de cobertura de sentencias (sustituye una actividad por la actividad `exit`).

CDE Aplica el criterio de cobertura de decisión (sustituye una decisión por *true()* o *false()*).

CCO Aplica el criterio de cobertura de condición (sustituye una condición por *true()* o *false()*).

CDC Aplica el criterio de cobertura de decisión / condición (sustituye una decisión o condición por *true()* o *false()*).

Además, se implementan varios operadores relacionados con la cobertura de código:

EIU Inserta el menos unario en expresiones aritméticas.

EIN Inserta la negación (la función *not()*) en expresiones lógicas.

EAP Inserta el valor absoluto positivo (la función *abs()*) en expresiones aritméticas.

EAN Inserta el valor absoluto negativo (la función *-abs()*) en expresiones aritméticas.

Además, es necesario disponer de, al menos, una composición a la que poder aplicar estos operadores, y de disponer de un conjunto de casos de prueba que cumpla los criterios de cobertura de código que aplican los operadores implementados.

El proceso llevado a cabo para realizar esta tarea puede resumirse en:

1. Implementar cada operador, escribiendo la hoja de estilo XSLT correspondiente, de manera que realice la mutación deseada.
2. Escribir el código Java correspondiente a las pruebas unitarias empleando el framework JUnit, atendiendo a los siguientes criterios:

   a) Detección del número de operandos en la composición.

Figura 5.14.: Representación del individuo

b) Cambios generados al aplicar los operadores.

c) Comprobación de que las salidas de la composición original y del mutante son diferentes.

3. Seleccionar una composición a la que poder aplicar los operadores implementados. En caso de no encontrar ninguna que se ajuste a las necesidades, adaptar una existente.

4. Comprobar que el conjunto de casos de prueba de la composición seleccionada cumple los criterios de cobertura de código. Si se ha adaptado una, crear un conjunto de casos de prueba que cumpla los criterios de cobertura de código.

5.5.3. Operadores de mutación

En la figura 5.14 podemos ver cómo se representan los individuos [29].

Operador Sirve para identificar al operador que será aplicado a la hora de realizar la mutación. Es un número entero comprendido entre 1 y el número de operadores definidos hasta el momento (en este caso, 34).

Operando Identifica la instrucción a la que será aplicado el operador. Es un número entero perteneciente al rango $[1,\ I]$, donde $I = mcm(m_i \mid 1 \leq i \leq 34)$, siendo m_i el número de instrucciones donde puede ser aplicado el i-ésimo operador. Esto se hace para hacer una distribución uniforme entre todos los individuos, sin tener en cuenta el número de instrucciones asociadas a cada uno. Siendo I_i el valor del campo instrucción en el individuo, el número de la instrucción a mutar será $\left\lceil \frac{I_i \cdot m_i}{I} \right\rceil$.

Atributo Representa la mutación que debe realizar el operador en el operando dado. Al igual que ocurría en el caso anterior, para obtener una distribución uniforme entre los individuos, este campo se codifica, siendo un número entero en el intervalo $[1,\ V]$, donde $V = mcm(v_i \mid 1 \leq i \leq 34)$, siendo v_i el número de posibles valores del atributo para el operador i-ésimo. Siendo $A_i$el valor del campo atributo para un individuo, su valor real será $\left\lceil \frac{A_i \cdot v_i}{V} \right\rceil$.
En la tabla 5.1 tenemos los valores máximos del atributo de todos los operadores definidos[1]. Podemos distinguir 3 tipos de operadores en función del uso que se haga del atributo:

[1]Cabe destacar que el valor máximo del atributo de los operadores ISV y XTF dependen del programa, ya que dependen del número de variables y de fallos, respectivamente.

Nombre	Número	Máximo	
ISV	1	N	
EAA	2	5	$+, -, *, div, mod$
EEU	3	1	
ERR	4	6	$<, >, \geq, \leq, =, \neq$
ELL	5	2	$\wedge, \vee$
ECC	6	2	$/, //$
ECN	7	4	$+1, -1, añadir, eliminar$
EMD	8	2	$0, mitad$
EMF	9	2	$0, mitad$
ACI	10	1	
AFP	11	1	
ASF	12	1	
AIS	13	1	
AIE	14	1	
AWR	15	1	
AJC	16	1	
ASI	17	1	

Nombre	Número	Máximo	
APM	18	1	
APA	19	1	
XMF	20	1	
XMC	21	1	
XMT	22	1	
XTF	23	N	
XER	24	1	
XEE	25	1	
AEL	26	1	
EIU	27	1	
EIN	28	1	
EAP	29	1	
EAN	30	1	
CFA	31	1	
CDE	32	2	$true(), false()$
CCO	33	2	$true(), false()$
CDC	34	2	$true(), false()$

Tabla 5.1.: Valores y atributos de los operadores de mutación

1. Operadores que intercambian elementos. A esta categoría pertenecen los operadores como CDE, el cual emplea el atributo para discernir cúando debe cambiar una decisión por *true()* o *false()*.

2. Operadores que cambian el orden de los elementos. Es el caso del operador ASI, el cual emplea el atributo para indicar si realiza el intercambio de instrucción delante o detrás de la actual.

3. Operadores cuyo atributo vale 1. Estos operadores realizan sólamente un tipo de mutación, por lo tanto no emplean el contenido de este campo, quedando por defecto a 1.

Como podemos apreciar, esta notación de los individuos es lo suficientemente flexible como para ser aplicada a cualquier lenguaje, ya que consiste únicamente en numerar los operadores y los posibles valores del atributo, en cada caso.

La implementación de los operadores de mutación se ha llevado a cabo empleando el lenguaje orientado a objetos Java, y el lenguaje de plantillas XSLT.

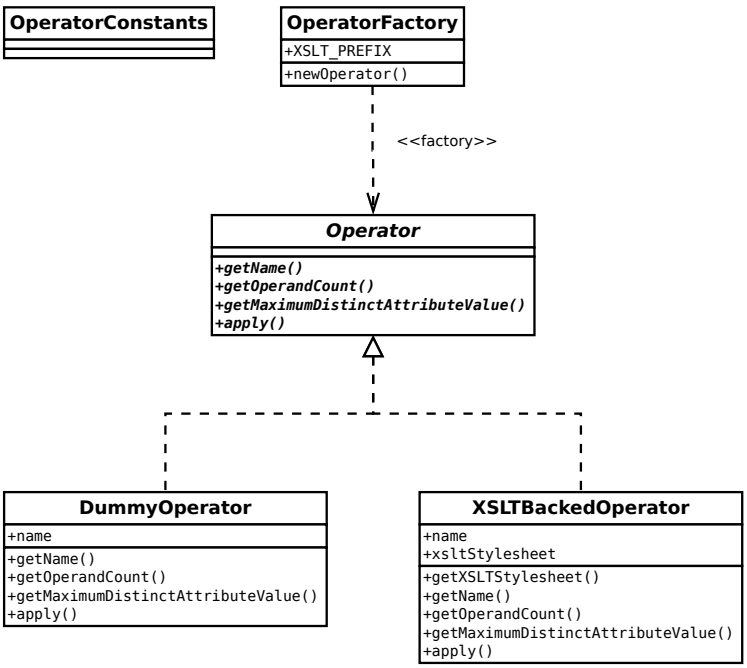

Figura 5.15.: Diagrama de clases de operadores

5.5.3.1. Java

El empleo del lenguaje Java está orientado hacia la mantenibilidad del sistema. Se busca con ello que el conjunto de operadores sea fácilmente ampliable, haciendo más fácil la tarea de incluir nuevos operadores en **MuBPEL**.

El patrón de diseño escogido es el patrón Fábrica [27], el cual permite abstraer al sistema de qué tipo de operador es necesario crear.

En la figura 5.15 podemos ver las clases que componen los operadores. La clase *OperatorConstants* contiene información relativa a los operadores. La interfaz *Operator* ofrece los métodos comunes para todos los operadores.

La clase OperatorFactory es la fábrica encargada de crear las instancias de los operadores. Su método `newOperator()` se encarga de buscar si existe la hoja XSLT correspondiente al operador en cuestión. En caso de encontrarla, crea una instancia de *XSLTBackedOperator*, un operador parametrizado con la hoja de estilo dada. En caso de no existir ésta, la fábrica crea una instancia de *DummyOperator*, la cual actúa como operador nulo, ya que no realiza mutación alguna.

5.5.3.2. XSLT

Se emplea el lenguaje de plantillas XSLT por su capacidad de transformar archivos XML a cualquier otro formato de texto de manera sencilla.

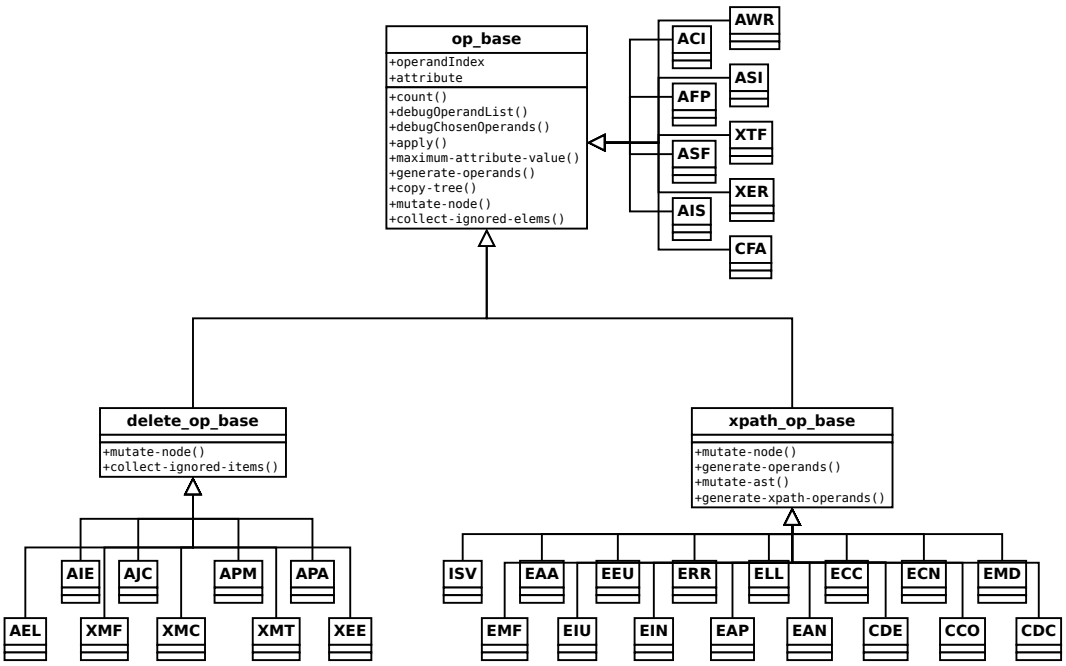

Figura 5.16.: Diagrama de hojas de estilo de operadores

Organización de las hojas de estilo

Las hojas de estilo residen en el directorio del proyecto `src/main/resources/es/uca/webservices/mutants/operators`. En la figura 5.16 podemos ver la jerarquía de las mismas. La estructura es la siguiente:

- `op-base.xsl`. Es la hoja de estilo en la que se basan todos los operadores de mutación.
- `delete-op-base.xsl`. Es la hoja de estilo en la que se basan todos los operadores de mutación que eliminan actividades.
- `xpath-op-base.xsl`. Es la hoja de estilo en la que se basan todos los operadores de mutación que afectan a expresiones XPath.
- `abc.xsl`. Las hojas de estilo con esta nomenclatura son las hojas propiamente dichas de los operadores de mutación.
- `generate-invalid-mutant.xsl`. Es una hoja de estilo especial para generar mutantes *stillborn*[2].

En función de la hoja de estilos de la que hereden los operadores, podemos apreciar que existen tres tipos de operadores:

- Los operadores que afectan a actividades (heredan de op-base).
- Los operadores que eliminan actividades (heredan de delete-op-base).

[2]Un mutante stillborn es aquel que no llega a desplegarse correctamente.

Composición	CFA	CDE	CCO	CDC	EIU	EIN	EAP	EAN
LoanApproval	✓	✓	✓	✓	✓	✓	✓	✓
MarketPlaceFlow	✓	✓	✓	✓	✗	✓	✗	✗
MetaSearch	✓	✓	✓	✓	✓	✓	✓	✓
SquaresSum	✓	✗	✗	✗	✓	✗	✓	✓
TacService	✓	✓	✓	✓	✓	✓	✓	✓
TravelReservationService	✓	✓	✓	✓	✓	✓	✓	✓

Tabla 5.2.: Operadores aplicables a cada composición

- Los operadores que afectan a expresiones XPath (heredan de xpath-op-base).

Los operadores de cobertura pertenecen a la última de las categorías, ya que afectan a expresiones XPath, con la excepción del operador de cobertura de sentencias, ya que éste afecta directamente a actividades.

5.5.4. Adaptación de una composición WS-BPEL

Para poder llevar a cabo la implementación de los operadores citados anteriormente, es necesario disponer al menos de una composición WS-BPEL en la que poder aplicarlos.

El operador CFA puede ser aplicado a cualquier composición WS-BPEL. Los operadores CDE, CCO y CDC necesitan una composición que tenga, al menos, una decisión. Los operadores EIU, EAP y EAN requieren una composición que tenga como mínimo una expresión aritmética para poder ser aplicados, y el operador EIN necesita la presencia de una expresión lógica, al menos.

El grupo UCASE dispone de una serie de composiciones de ejemplo, las cuales están disponibles en: https://neptuno.uca.es/redmine/projects/sources-fm/repository/show/trunk/samples. En la tabla 5.2 podemos ver los operadores que son aplicables a las composiciones escogidas.

LoanApproval Es una composición que implementa un servicio de aprobación de préstamos bancarios, mediante un asesor y un aprobador.

MarketPlaceFlow Esta composición modela un servicio de compra-venta, con un comprador y un vendedor. Es un ejemplo donde ambas entidades entran en juego concurrentemente, así que sirve como ejemplo de sincronización entre servicios concurrentes. Procede de la web de ActiveVOS [4].

MetaSearch Implementa un buscador de Internet, en el que el cliente solicita una búsqueda, la cual devuelve el motor. Procede de BPELUnit [1].

SquaresSum Es una composición que calcula $\sum_{i=0}^{n} i^2$, empleando un bucle `<forEach>` paralelo.

TacService Imita a la orden de UNIX `tac`, la cual se encarga de invertir las líneas de un fichero dado.

TravelReservationService Ejemplo de una agencia de viajes, con servicio de reserva de vuelo, hotel y coche de alquiler. Procede de la web de NetBeans [33], pero ha sido adaptado a las necesidades de los investigadores del grupo UCASE.

Se estudiaron las ventajas e inconvenientes del empleo de cada una de las composiciones a las que se le podían aplicar todos los operadores. La composición *MetaSearch* fue descartada por el hecho de que estuvo dando problemas mientras otros miembros del grupo investigaban con ella. La composición *tacService* no cumplía los mínimos de complejidad que necesitan estos operadores. La composición *TravelReservationService*, a pesar de ser la más empleada, tampoco es la más indicada, ya que no tiene las características que se necesitan.

La composición *LoanApproval* era la que más se acercaba al cumplimiento de los requisitos, así que se decidió adaptarla, de manera que los operadores implementados pudieran aplicarse, pudiendo apreciarse las diferencias entre los diferentes criterios de cobertura.

Composición LoanApproval original

En la figura 5.17 podemos apreciar el comportamiento de la composición LoanApproval original. Como vemos, el cliente solicita un préstamo de una cantidad determinada. Si la cantidad es inferior o igual a 10000, el asesor se encarga de determinar el riesgo de la operación. En función del riesgo, se aprueba directamente el préstamo o se deja a decisión del aprobador. Si la cantidad supera los 10000, decide directamente el aprobador sin tener en cuenta el riesgo de la operación.

Vemos que esta composición consta de 2 decisiones, las cuales son atómicas, ya que sólo están compuestas de una condición cada una. Además, esta composición no consta de expresiones aritméticas complejas. Así que para solventar estas limitaciones, se introducen las modificaciones que se explican a continuación[3]:

Adaptación de la composición para los operadores CDE, CCO y CDC

Se añade una condición nueva en la primera decisión, para que los estudios de cobertura de decisión, condición y decisión / condición no sean idénticos, comprobando que la cantidad solicitada oscile entre 1000 y 10000.

[3]No es necesario adaptar la composición para aplicar los operadores CFA y EIN, ya que existen suficientes actividades y expresiones lógicas en la misma.

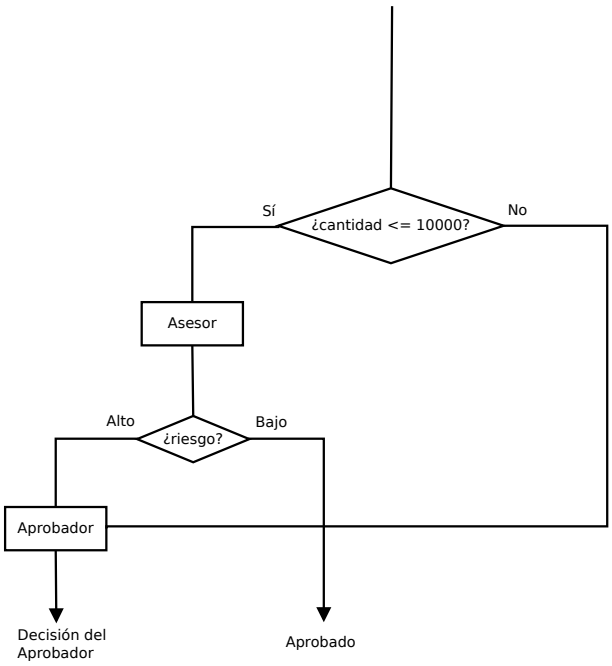

Figura 5.17.: Diagrama de flujo de LoanApproval

Adaptación de la composición para los operadores EIU, EAP y EAN

La modificación anterior no cubría el caso en que el préstamo fuese inferior a 1000, en ese caso, se añade una nueva decisión, para comprobar ese hecho, y en caso de ocurrir, se invoca al asesor, para comprobar el riesgo de la operación. En caso de que el riesgo sea bajo, el asesor propone una cantidad nueva, la cual será un $10\,\%$ mayor a la solicitada por el cliente. Esto permite la introducción de una expresión aritmética compleja, a la que se podrán aplicar los operadores EIU, EAP y EAN.

En la figura 5.18 podemos ver el diagrama de flujo de la composición LoanApproval modificada.

5.6. Implementación

5.6.1. Sistema de ejecución de GAmera

En la figura 5.19 se distinguen los pasos que sigue GAmera a la hora de ejecutar tanto la composición WS-BPEL original como los mutantes.

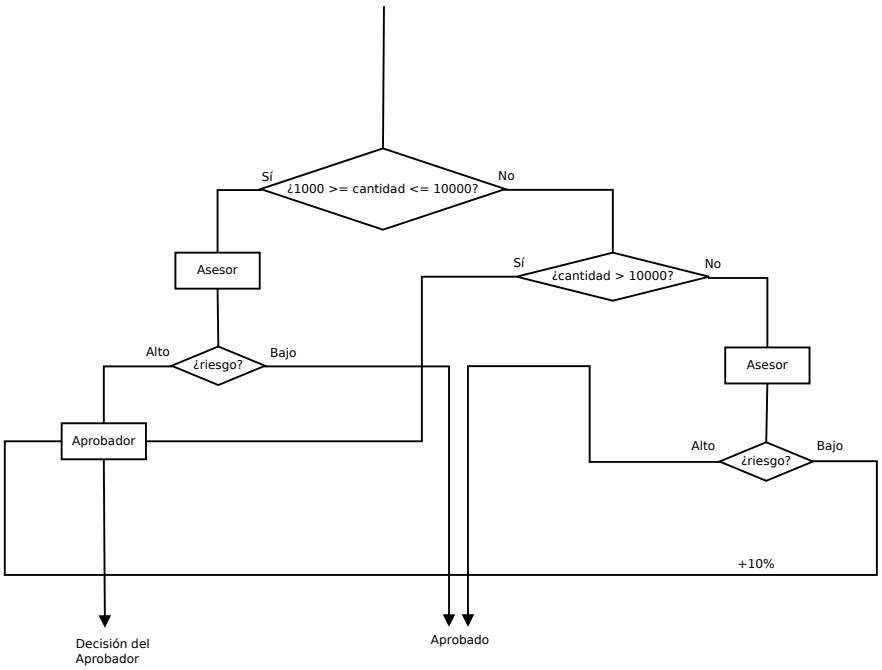

Figura 5.18.: Diagrama de flujo de LoanApproval modificada

Empaquetado

En este paso, se empaqueta la composición WS-BPEL y se prepara para su ejecución en el motor ActiveBPEL [4]. Se generan los siguientes ficheros necesarios en la ejecución:

- `catalog.xml`. Fichero que contiene las dependencias del proceso WS-BPEL.
- `process.pdd`. Fichero de configuración para el despliegue del proceso en ActiveBPEL.
- `types.xml`. Fichero con las declaraciones de tipos de los XML y WSDL.

Al final de este paso, se genera un fichero con extensión bpr, el cual contiene los ficheros anteriores, además de la composición original, los ficheros WSDL correspondientes a los servicios web y el conjunto de casos de prueba.

Ejecución

Una vez empaquetado el mutante, comienza el proceso de ejecución, que puede resumirse en:

1. Se despliega la composición WS-BPEL en ActiveBPEL, enviando el archivo con extensión bpr generado durante el empaquetado.

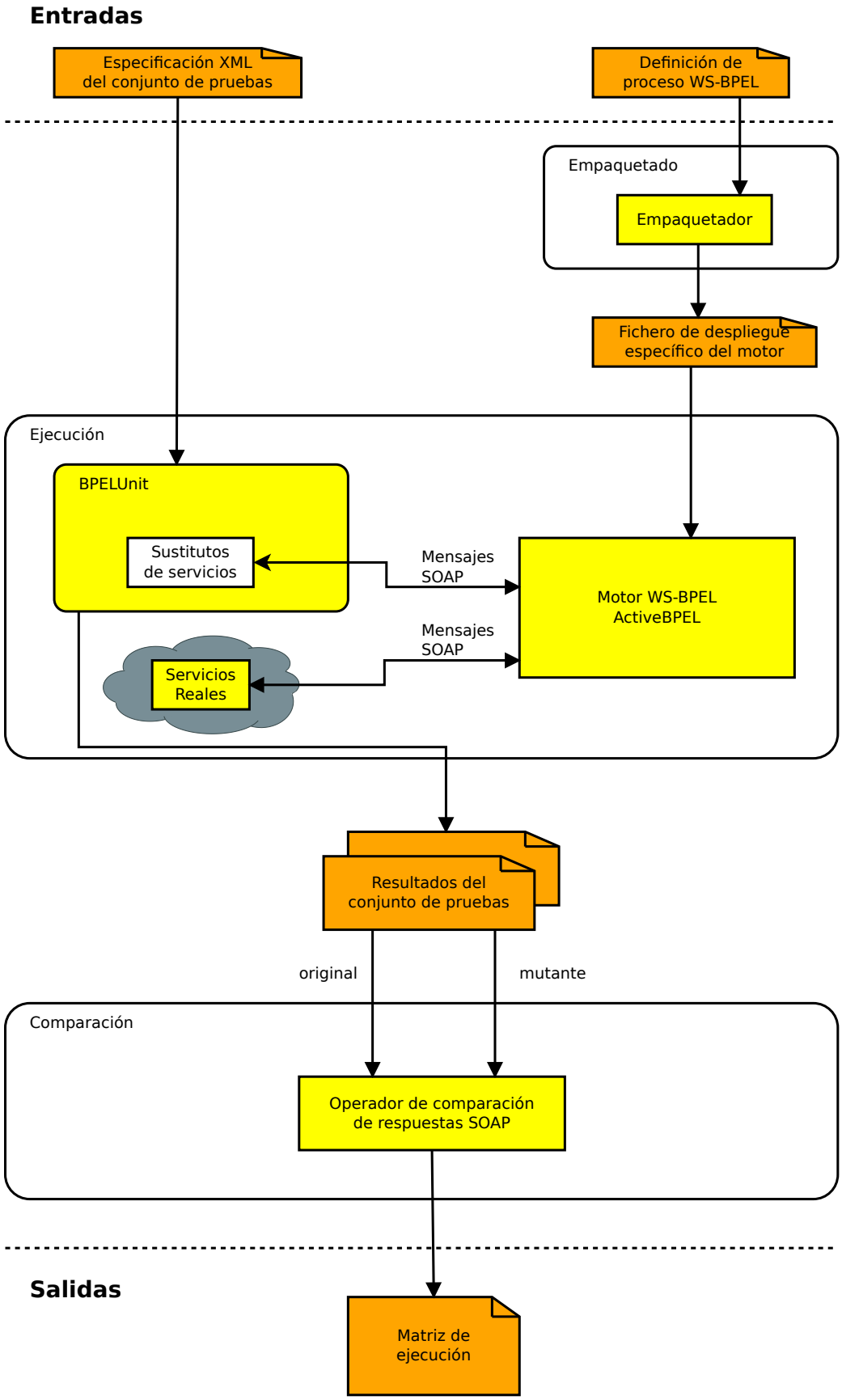

Figura 5.19.: Sistema de ejecución de GAmera

2. Se levanta el servidor que se encarga de simular las llamadas a servicios web externos.

3. En cada caso de prueba, se realiza lo siguiente:

   a) Se configura el *mockup* para que responda de la manera que se desee.

   b) Se invoca el proceso WS-BPEL.

   c) Se generan los resultados de la ejecución.

4. Una vez ejecutados todos los casos de prueba, se detiene el proceso WS-BPEL.

El encargado de dirigir el proceso de ejecución es BPELUnit. A partir del fichero bpr, se encarga de configurar los *mockups* con las respuestas predefinidas adecuadas, y sustituye en la composición las llamadas a servicios reales por las llamadas a los servicios simulados. Los principales motivos de no emplear los servicios reales residen en la reducción de costes y en la intención de mantener el control sobre el entorno de prueba.

BPELUnit se encarga además de realizar la invocación de los servicios, actuando como cliente en la ejecución de la composición durante las pruebas.

Comparación

Una vez termina la ejecución, se comparan las salidas de los mutantes con la de la composición original, comparando una a una las respuestas SOAP producidas durante la ejecución. Como resultado se obtiene la matriz de ejecución. En función del estado del mutante, los elementos de la matriz pueden tener uno de los valores siguientes:

- $m_{ij} = 0$. Esto ocurre cuando la salida del mutante i coincide con la de la composición original en el caso de prueba j.
- $m_{ij} = 1$. En este caso, la salida del mutante i es distinta a la de la composición original para el caso de prueba j.
- $m_{ij} = 2$. Este valor se debe a que existen errores en la ejecución del mutante i.

Sean M el número de mutantes, T el número de casos de prueba del conjunto y $A \in \mathcal{M}_{M \times T}(m_{ij})$.

- Si $\sum_{j=1}^{T} m_{ij} = 0$, el mutante i está vivo.
- Si $\sum_{j=1}^{T} m_{ij} \geq 1$, el mutante i está muerto.
- Si $\sum_{j=1}^{T} m_{ij} = 2T$, el mutante i es erróneo.

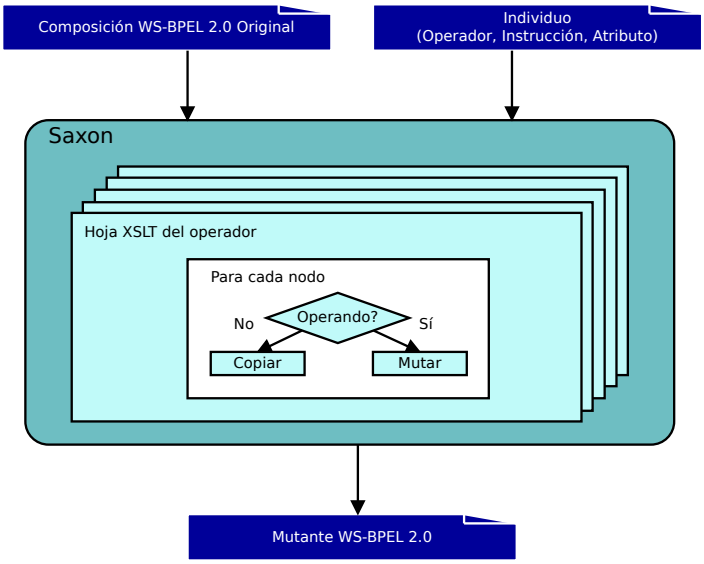

Figura 5.20.: Conversión de individuo a mutante

5.6.2. Conversión individuo-mutante

La conversión de individuo a mutante se produce aplicando la hoja de estilo correspondiente al operador, como podemos apreciar en la figura 5.20.

Se tiene la composición WS-BPEL original y el individuo con el operador, el operando y el atributo. El motor Saxon recorre los nodos del árbol de la composición original, buscando el nodo operando. En caso de que el nodo tratado actualmente sea el operando, aplica la mutación indicada en la plantilla. En caso contrario, copia el nodo actual. El producto obtenido al final de este proceso es un mutante de la composición WS-BPEL original.

5.6.3. Implementación de operadores

El motor XSLT empleado ha sido Saxon 9.1-B. A continuación se describen las 3 hojas principales en las que se basan los operadores de mutación.

5.6.3.1. op-base.xsl

Esta hoja de estilos posee los atributos y métodos comunes que todos los operadores necesitan. Los atributos que implementa son:

- `operandIndex`: Valor numérico de la instrucción que va a sufrir la mutación.
- `attribute`: Valor del atributo en la mutación.

Los métodos que ofrece son:

- `count()`: Cuenta el número de operandos obtenidos tras el análisis, es decir, el número de instrucciones que pueden ser mutadas.
- `debugOperandList()`: Ayuda a la depuración de la lista de operandos de los operadores.
- `debugChosenOperands()`: Ayuda a la depuración de los operandos escogidos.
- `apply()`: Aplica la mutación sobre el operando elegido.
- `maximum-attribute-value()`: Devuelve el valor máximo del atributo.

Las plantillas que ofrece son:

- `generate-operands()`: Obtiene la lista de operandos donde se puede aplicar el operador. Las hojas de estilo derivadas deben definir esta plantilla en función de los operandos en los que deban de aplicarse los operadores.
- `copy-tree()`: Copia el árbol DOM del documento WS-BPEL. En el caso de requerir alguna acción especial durante la copia, es necesario redefinir esta plantilla en la hoja derivada. Esta plantilla aplica las plantillas con modo `mutate-node`, las cuales se encargan de aplicar la mutación correspondiente a los nodos del árbol.
- `collect-ignored-items()`: Recoge los elementos ignorados al copiar, su comportamiento por defecto es que no se ignore nodo alguno. Por tanto, al igual que en las anteriores, si queremos un comportamiento distinto, redefiniremos el comportamiento de la misma cuando sea necesario.

5.6.3.2. delete-op-base.xsl

Esta hoja de estilos, derivada de `op-base.xsl`, define las características que necesitan los operadores que eliminen elementos.

Define la plantilla con modo `mutate-node`, dándole el comportamiento por defecto, que es el de eliminar el nodo del árbol.

Además, define la plantilla `collect-ignored-items()` de manera que se eliminen los enlaces al eliminar los elementos del árbol.
Atendiendo al tipo de enlace, se pueden dar 3 casos:

1. El enlace está declarado y definido dentro del operando.
   En este caso, no es necesario hacer nada especial, sólo realizar la eliminación.
2. El enlace está declarado fuera del operando y definido dentro del mismo.
   En este caso, es necesario eliminar el enlace además del operando.

3. El enlace está declarado y definido fuera del operando.
   En este caso, es necesario recorrer el árbol para eliminar el enlace, además de eliminar el operando.

Por tanto, los operadores que deban eliminar elementos sólo deberán definir la plantilla `generate-operands()`.

5.6.3.3. xpath-op-base.xsl

Esta hoja de estilos, derivada de `op_base.xsl`, es la empleada como base para los operadores que afecten a expresiones XPath.

Ofrece la plantilla `generate-xpath-operands()`, en la que se seleccionan los operandos dentro del AST de todas las expresiones XPath, a través de una consulta XPath, para lo cual se emplea la función `saxon:evaluate()`.

Define la plantilla con modo `mutate-node`, de tal manera que se recorre el AST de la expresión XPath, aplicando las plantillas con modo `mutate-ast`, encargadas de mutar el operando dado.

Así pues, los operadores que muten expresiones XPath sólo deben de definir las plantillas `generate-xpath-operands()` y la que tenga modo `mutate-ast`.

5.6.3.4. Lista de operandos y de ignorados

Todas las hojas XSLT generan una lista de operandos mediante su plantilla `generate-operands()` o `generate-xpath-operands()`, dependiendo si se mutan expresiones XPath o no.

Esta lista se emplea a la hora de aplicar el operador al operando, además de en las opciones de depuración de los operandos. Esta lista es heterogénea, y está separada por elementos WS-BPEL. No se puede anidar, ya que si se anidase se copiarían los nodos, con lo que no se podría emplear el operador de XPath `is` para ver si el nodo actual es el que se desea mutar. Cada operando en la lista está compuesto por:

- `BPELNode`: Nodo del árbol DOM WS-BPEL que constituye el operando.
- `ast`: Si procede, elemento raíz del AST de la expresión XPath afectada. Es un elemento `uca:expression`.
- `extra`: Se emplea para guardar información adicional acerca del operando. Cuando se mutan expresiones XPath, se emplea para guardar el nodo del AST que forma parte del operando. En los operadores ASI, ISV y XTF se introduce el índice.

La lista de ignorados se genera mediante la plantilla `collect-ignored-items()`, y es empleada a la hora de copiar el árbol, para que se evite la copia de los elementos que estén presentes en la misma.

5.6.3.5. Implementación del operador CDC

En el listado 5.1 podemos ver la implementación del operador CDC. El resto de operadores están disponibles en el CD adjunto.

Listado 5.1: Implementación del operador CDC

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!--
3     CDC (Coverage: Decision/Condition) WS-BPEL mutation operator
4
5     Applies the decision/condition coverage criteria to a WS-BPEL composition.
6
7     Valentin Lineiro Barea <valentin.lineirobarea@alum.uca.es>
8  -->
9  <xsl:stylesheet
10    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
11    xmlns:xs="http://www.w3.org/2001/XMLSchema"
12    xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
13    xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"
14    xmlns:uca="http://www.uca.es/xpath/2007/11"
15    xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
16    version="2.0">
17
18  <xsl:import href="es/uca/webservices/mutants/operators/xpath-op-base.xsl"/>
19
20  <xsl:param name="action" as="xs:string"/>
21  <xsl:param name="operandIndex" as="xs:integer"/>
22  <xsl:param name="attribute" as="xs:integer"/>
23
24  <xsl:variable name="fixedMaximumAttributeValue" select="2"/>
25
26  <xsl:template name="generate-operands">
27    <xsl:call-template name="generate-xpath-operands">
28      <!--
29            Selects all decisions and conditions that exists into
30            the composition.
31       -->
32      <xsl:with-param name="selectedNodes"
33      select="concat('uca:expression/(uca:eq | uca:neq | uca:gt | uca:ge |
           uca:lt | uca:le |
34     uca:and | uca:or | uca:variableReference | uca:functionCall/uca:qname[
           @localPart = ', $quot, 'not',
35     $quot, ']/(following-sibling::uca:eq | following-sibling::uca:neq |
36     following-sibling::uca:gt | following-sibling::uca:ge |
37     following-sibling::uca:lt | following-sibling::uca:le |
38     following-sibling::uca:and | following-sibling::uca:or)) |
```

```
39      uca:and/(uca:gt | uca:ge | uca:lt | uca:le | uca:eq | uca:neq) |
40      uca:or/(uca:gt | uca:ge | uca:lt | uca:le | uca:eq | uca:neq) |
41      (uca:and | uca:or)/uca:variableReference |
42      uca:functionCall/uca:qname[@localPart = ' , $quot, 'not', $quot, ']/(
            following-sibling::uca:eq |
43      following-sibling::uca:neq | following-sibling::uca:gt | following-
            sibling::uca:ge |
44      following-sibling::uca:lt | following-sibling::uca:le)')"/>
45      <xsl:with-param
46          name="selectedBPELNodes"
47          select="'//bpel:condition | //bpel:joinCondition |
48                      //bpel:transitionCondition'"/>
49    </xsl:call-template>
50  </xsl:template>
51
52  <xsl:template match="*" mode="mutate-ast-node">
53    <xsl:choose>
54    <!--
55                Uses the attribute variable to select when
56                mutate with true() or false().
57     -->
58          <xsl:when test="$attribute = 1">
59            <uca:functionCall>
60          <uca:qname localPart="true" prefix=""/>
61        </uca:functionCall>
62          </xsl:when>
63       <xsl:otherwise>
64            <uca:functionCall>
65          <uca:qname localPart="false" prefix=""/>
66        </uca:functionCall>
67          </xsl:otherwise>
68        </xsl:choose>
69  </xsl:template>
70
71  </xsl:stylesheet>
```

El documento comienza con la declaración XML, en la que se indica el empleo de la versión 1.0 (línea 1). A continuación, aparece el elemento raíz del mismo, el elemento `xsl:stylesheet`. Aparecen los espacios de nombres empleados dentro del documento y la versión del lenguaje empleada, XSLT 2.0 (líneas 9 a 16).

En la línea 18 se importa la hoja de estilos `op-base.xsl` a través del elemento `xsl:import`. Mediante este elemento podemos importar otras hojas de estilo a la nuestra, simulando el mencanismo de la herencia que poseen los lenguajes orientados a objetos, ya que las plantillas y funciones importadas tienen menos prioridad que las definidas en nuestra hoja, lo que permite redefinirlas para ajustarlas a nuestras necesidades.

En las líneas 20 a 22 se declaran los parámetros globales de la hoja de estilos, la acción que queremos tomar (`action`), el número de la instrucción que deseamos

mutar (`operandIndex`) y el valor del atributo (`attribute`).

En la línea 24, se cambia el valor máximo del atributo, cuyo valor por defecto es 1 en `op-base.xsl`, redefiniendo la variable `fixedMaximumAttributeValue` y dándole valor 2, ya que el operador CDC realiza dos tipos de cambios, *true()* o *false()*.

A continuación se llama a la plantilla `generate-operands`, la cual genera la lista de operandos donde se puede aplicar el operador en la composición dada (líneas 26 a 50). Esta plantilla se hereda de la hoja `xpath-op-base.xsl`. En este caso, se seleccionan en primer lugar todas las decisiones y a continuación las condiciones, teniendo cuidado con las que están negadas, ya que forman parte de la llamada a la función *not()* de XPath y teniendo en cuenta las variables booleanas, que sean hijas de un elemento *and* u *or*. Además, se especifican los parámetros con los que se llama a la plantilla, especificando que seleccione los elementos WS-BPEL que sean condiciones, es decir, los elementos `<condition>`, `<joinCondition>`, y `<transitionCondition>`, excluyendo las expresiones aritméticas.

Por último, se define la plantilla `mutate-node`, la plantilla que especifica la mutación que debe realizar el operador (líneas 52 a 69). Esta plantilla se emplea automáticamente con todos los elementos BPEL, al concordar con *, excluyendo nodos texto y nodos atributo. En este caso, si el valor del atributo es 1, sustituimos por una llamada a la función *true()*, en otro caso, sustituimos por una llamada a la función *false()*.

5.7. Pruebas

5.7.1. Plan de pruebas

5.7.1.1. Alcance

Se han realizado pruebas a los operadores de mutación, a los resultados obtenidos tras ejecutar la composición *LoanApproval* modificada y sus mutantes, y a los resultados obtenidos tras comparar las salidas de la composición *LoanApproval* modificada y de sus mutantes.

5.7.1.2. Tiempo y lugar

Las pruebas se han llevado a cabo mientras se desarrollaba el proyecto. Una vez finalizada la implementación de un operador, se ejecutaban sus pruebas correspondientes. Si las pruebas tenían éxito, se pasaba a la implementación de un nuevo operador. En caso contrario, se depuraban los errores y se volvían a ejecutar las pruebas.

Una vez finalizados todos los operadores, se ejecutaron las pruebas de manera global, con el objeto de comprobar que no existían errores posteriores.

5.7.1.3. Naturaleza

Todas las pruebas realizadas han sido estructurales (de caja blanca). Recordemos que las pruebas estructurales se centran en comprobar que la estructura interna del software es la adecuada, al contrario que las pruebas de caja negra, que se encargan de verificar el dominio de las entradas y salidas del software a probar.

5.7.2. Pruebas unitarias para los operadores

Para crear las pruebas unitarias de los operadores, se ha empleado el framework JUnit [39]. Este framework aporta una serie de clases que sirven para implementar de manera cómoda el conjunto de casos de prueba necesario empleando el lenguaje Java.

Se especializa la fábrica *OperatorFactory*, creando una fábrica *TestOperatorFactory*, la cual genera operadores especiales para las pruebas.

Se especializa la clase *XSLTStylesheet*, creando la clase *TestXSLTStylesheet*, la cual se encarga de recoger todos los errores generados por la hoja XSLT en la transformación actual, almacenando los errores, errores fatales y avisos en listas. Además, los errores y errores fatales son lanzados como excepciones de la clase *TransformerException*.

Para la prueba de los operadores en las composiciones, se implementa la clase *BPELMutationTestCase*. Esta clase incorpora los métodos y atributos necesarios para definir las pruebas unitarias de cada una de las composiciones WS-BPEL. Así pues, todas las composiciones deberán crear una clase que herede de ésta, para realizar las pruebas. Los casos de prueba definidos en las composiciones atienden al siguiente criterio:

1. Detección del número de operandos en la composición.
2. Cambios generados al aplicar los operadores.
3. Comprobación de que las salidas de la composición original y del mutante son diferentes.

5.7.2.1. Pruebas unitarias para el operador CDC

En el listado 5.2 se adjuntan las pruebas unitarias realizadas al operador CDC. En el CD adjunto pueden consultarse el resto de pruebas unitarias realizadas.

Listado 5.2: Pruebas unitarias del operador CDC

```
1  public class LoanApprovalProcessCoverageTest extends BPELMutationTestCase {
2
3      private final static String TEST_SUITE = "loanCoverage/LoanApprovalProcess.bpts"
           ;
4
5      public LoanApprovalProcessCoverageTest() {
6        super("loanCoverage/LoanApprovalProcess.bpel");
7      }
8
9      /* CDC TESTS */
10
11     @Test
12     public void cdcOperandCount() throws Exception {
13       assertOperandCount(6, getOp("CDC"));
14     }
15
16     @Test
17     public void cdcAttributeRange() throws Exception {
18         assertMaximumDistinctAttributeValueIs(getOp("CDC"), 2);
19     }
20
21     @Test
22     public void cdcChangeIf2False() throws Exception {
23         applyAndAssertEqualsXPath(
24                                   getOp("CDC"),
25                                   2,
26                                   1,
27                                   "//bpel:if[@name='If1']/bpel:condition/text()",
28                                   "(true() and (number(string($processInput.input/
                                         ns0:amount)) >= 1000.0))");
29     }
30
31     @Test
32     public void cdcMutatesIf2False() throws Exception {
33       assertSuccessfulMutationIsDifferent(TEST_SUITE, getOp("CDC"), 2, 1);
34     }
35
36 }
```

En la línea 1 podemos ver que se crea una clase *LoanApprovalProcessCoverageTest* que extiende a *BPELMutationTestCase*. Más adelante se indica el *path* hacia la composición WS-BPEL (línea 3).

El constructor de la clase (línea 5) inicializa la instancia de la superclase con el *path* indicado. Una vez terminada la parte de configuración, aparecen las pruebas unitarias (líneas 11-34).

La primera de las pruebas, `cdcOperandCount()` (línea 11), verifica que para la composición *MetaSearch* el número de operandos para el operador CDC es 6. Sabemos que es así porque previamente se han contado manualmente en la composición.Por tanto,

si a la hora de analizar no hubiese 6 operandos, el operador no estaría implementado correctamente.

La segunda de las pruebas, `cdcAttributeRange()` (línea 16), comprueba que el valor máximo del atributo para este operador es 2. Este operador realiza dos tipos de mutaciones, cambiando la condición o la decisión por *true()* o *false()*, así pues, debemos obtener ese valor en la prueba.

La tercera de las pruebas, `cdcChangeIf2False()` (línea 21), se encarga de ver si la mutación realizada por el operador CDC en el 2º operando, con valor de atributo 1 es la que aparece descrita.

La última de las pruebas, `cdcMutatesIf2False()` (línea 31), verifica que la salida de la composición original es distinta de la salida del mutante generado tras aplicar CDC en el 2º operando con valor de atributo 1. Cabe esperar que sea así, ya que altera el valor de la condición en la que se verifica el límite superior de la cantidad, dejándola por defecto a verdadero. En los casos de prueba en los que no sea así, el mutante muere y las salidas difieren.

5.7.3. Casos de prueba para la composición WS-BPEL

Para crear los casos de prueba para la composición WS-BPEL se emplea el framework BPELUnit [1]. El conjunto de casos de prueba se define en un fichero con extensión *.bpts*, un fichero XML dependiente de la composición. En la figura podemos ver la estructura de un fichero *.bpts*.

Listado 5.3: Estructura de un fichero .bpts

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tes:testSuite
3      xmlns:esq="http://xml.netbeans.org/schema/Loans"
4      xmlns:ap="http://j2ee.netbeans.org/wsdl/ApprovalService"
5      xmlns:as="http://j2ee.netbeans.org/wsdl/AssessorService"
6      xmlns:sp="http://j2ee.netbeans.org/wsdl/LoanService"
7      xmlns:tes="http://www.bpelunit.org/schema/testSuite">
8
9    <tes:name>LoanServiceTest</tes:name>
10   <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
11
12   <tes:deployment>
13     <tes:put name="LoanApprovalProcess" type="activebpel">
14       <tes:wsdl>LoanService.wsdl</tes:wsdl>
15       <tes:property name="BPRFile">LoanApprovalDoc.bpr</tes:property>
16     </tes:put>
17     <tes:partner name="assessor" wsdl="AssessorService.wsdl"/>
18     <tes:partner name="approver" wsdl="ApprovalService.wsdl"/>
19   </tes:deployment>
20
21   <tes:testCases>
```

```
22      <tes:testCase name="Case1" basedOn="" abstract="false" vary="false">
23
24      </tes:testCase>
25
26      ...
27
28    </tes:testCases>
29  </tes:testSuite>
```

En la línea 2 podemos ver el elemento raíz, tes:testSuite. En él se declaran los espacios de nombres empleados dentro del documento XML. A continuación aparecen una serie de elementos, los cuales sirven para la configuración de los casos de prueba (líneas 9-19). Son:

- `tes:name`. En él se indica el nombre del proceso WS-BPEL.
- `tes:baseUrl`. En él se especifica la URL sobre la que el proceso será desplegado.
- `tes:deployment`. Configuración general del despliegue del proceso. Se configura lo siguiente:
  - `tes:put`. Se indica el proceso a desplegar, especificando el motor WS-BPEL, el fichero WSDL de descripción del servicio web y el paquete `bpr` con los ficheros necesarios.
  - `tes:partner`. Se indican los servicios web externos que necesita la composición durante su ejecución.

A partir de la línea 21, se especifican los casos de prueba (elemento `<tes:testCases>`). Éstos poseen la siguiente estructura:

- `tes:testCase`. Este elemento sirve para definir un caso de prueba. El atributo `name` identifica al caso de prueba. Un caso de prueba puede basarse en otro, indicado mediante `basedon`. Un caso de prueba puede ser abstracto, si el atributo `abstract` tiene el valor *yes*. Si eso ocurre, no puede ser ejecutado. Además, se puede indicar mediante el atributo `vary`, si es necesario repetir su ejecución variando algunos parámetros. Es necesaria la presencia de un elemento `tes:clientTrack` y de uno o varios elementos `partnerTrack` dentro de cada caso de prueba.
  - `tes:clientTrack`. Contiene la petición del cliente.
  - `tes:partnerTrack`. Define el *mockup* empleado, el cual se indica mediante el atributo `name`. Debe haber un elemento `tes:partnerTrack` por cada servicio web utilizado.

Los elementos `clientTrack` y `partnerTrack` pueden contener las siguientes actividades:

- `tes:sendOnly`. Mediante esta actividad se envía un mensaje simple.
- `tes:receiveOnly`. Mediante esta actividad se recibe y verifica un mensaje simple.
- `tes:sendReceive`. Mediante esta actividad, se envía una petición síncrona. Se compone de:
  - `tes:send`. A través de esta actividad, se envían los datos oportunos, dentro de un bloque tes:data. El atributo `fault` indica si se espera un fallo o no.
  - `tes:receive`. A través de esta actividad, se recibe la respuesta. Mediante elementos `tes:condition` podemos verificar que la respuesta recibida es la esperada.
- `tes:receiveSend`. Esta actividad espera una respuesta simple, tras la cual envía un mensaje síncrono.
- `tes:sendReceiveAsynchronous`. Mediante esta actividad se envía un mensaje simple, tras el cual se espera una respuesta asíncrona. Una vez que llega, se comprueban los datos recibidos.
- `tes:receiveSendAsynchronous`. Esta actividad espera la recepción de los datos, respondiendo luego de manera asíncrona.

A continuación, se mostrarán los casos de prueba que matan a los mutantes no equivalentes generados tras la aplicación de los operadores de mutación implementados.

5.7.3.1. Caso de prueba SmallAmountHighRisk

En la figura 5.4 tenemos disponible la implementación de este caso de prueba. Como podemos ver, este caso de prueba no se basa en ninguno, no es abstracto y no se realizan variaciones de parámetros (línea 1).

Más adelante aparece un elemento `clientTrack` (líneas 2-23). Dentro de este elemento, aparece una actividad `sendReceive`, la cual se encarga de enviar la petición del préstamo al servicio. En el mensaje de envío (elemento `send`), aparecen la cantidad solicitada, *5000*, dentro de un elemento data. Como respuesta (elemento `receive`), esperamos que el préstamos sea aprobado, lo que podemos ver dentro del elemento `condition`.

Una vez definida la petición del cliente, aparecen los *mockups* de los servicios del asesor financiero y del aprobador. El primer elemento `partnerTrack` (líneas 25-39), se encarga de definir las respuestas del aprobador en este caso de prueba. Se emplea la actividad `receiveSend`, mediante la cual se espera una petición de aprobación de préstamo. Una vez recibida, el aprobador responde aceptando el préstamo, lo que se indica en el elemento `data` del envío (elemento `send`).

El segundo elemento `partnerTrack` (líneas 41-57) representa al servicio del asesor financiero. Éste emplea una actividad `receiveSend`, al igual que ocurría con el aprobador, esperando la petición de asesoramiento. Una vez recibida, el asesor responde indicando que el riesgo de la operación bancaria es alto.

Listado 5.4: Caso de prueba SmallAmountHighRisk

```
1   <tes:testCase name="SmallAmountHighRisk" basedOn="" abstract="false" vary="
        false">
2     <tes:clientTrack>
3       <tes:sendReceive
4           service="sp:LoanServiceService"
5           port="LoanServicePort"
6           operation="grantLoan">
7
8         <tes:send fault="false">
9           <tes:data>
10            <esq:ApprovalRequest>
11              <esq:amount>5000</esq:amount>
12            </esq:ApprovalRequest>
13          </tes:data>
14        </tes:send>
15
16        <tes:receive  fault="false">
17          <tes:condition>
18            <tes:expression>esq:ApprovalResponse/esq:accept</tes:expression>
19            <tes:value>'true'</tes:value>
20          </tes:condition>
21        </tes:receive>
22      </tes:sendReceive>
23    </tes:clientTrack>
24
25    <tes:partnerTrack name="approver">
26      <tes:receiveSend
27          service="ap:ApprovalServiceService"
28          port="ApprovalServicePort"
29          operation="approveLoan">
30        <tes:send fault="false">
31          <tes:data>
32            <esq:ApprovalResponse>
33              <esq:accept>true</esq:accept>
34            </esq:ApprovalResponse>
35          </tes:data>
36        </tes:send>
37        <tes:receive  fault="false"/>
38      </tes:receiveSend>
39    </tes:partnerTrack>
40
41    <tes:partnerTrack name="assessor">
42      <tes:receiveSend
43          service="as:AssessorServiceService"
44          port="AssessorServicePort"
45          operation="assessLoan">
```

```
46
47          <tes:receive fault="false"/>
48          <tes:send fault="false">
49            <tes:data>
50              <esq:AssessorResponse>
51                <esq:risk>high</esq:risk>
52              </esq:AssessorResponse>
53            </tes:data>
54          </tes:send>
55
56        </tes:receiveSend>
57      </tes:partnerTrack>
58
59    </tes:testCase>
```

5.7.3.2. Caso de prueba SmallAmountHighRiskRejected

El código de este caso de prueba y de los que le siguen está disponible en el CD que se adjunta.

Este caso de prueba tiene idéntica estructura que el anterior. La diferencia reside en la decisión del aprobador, que en este caso, decide rechazar la petición de préstamo.

5.7.3.3. Caso de prueba SmallAmountLowRisk

El `clientTrack` de este caso de prueba es idéntico al de los dos anteriores. Sin embargo, el asesor en este caso considera que el riesgo de la operación es bajo, por lo tanto se procede a conceder directamente el préstamo. En este caso de prueba, el aprobador no interviene en ningún momento.

5.7.3.4. Caso de prueba LargeAmount

En este caso de prueba, el cliente realiza una petición de préstamo de 20000, valor que cambia dentro de los datos de la petición.

Dada la cantidad solicitada, el aprobador decide conceder directamente el préstamo, sin emplear el servicio de asesoramiento, el cual no interviene en ningún momento.

5.7.3.5. Caso de prueba LargeAmountRejected

Este caso de prueba es análogo al anterior, sólo que el aprobador decide no conceder el préstamo. La cantidad solicitada no varía, y tampoco interviene el asesor financiero.

5.7.3.6. Caso de prueba VerySmallAmountLowRisk

En este caso de prueba el cliente solicita 500 de préstamo. El asesor financiero, dada la cantidad, y tras determinar que el riesgo es bajo, decide realizar una oferta, ampliando el préstamo un 10 %, hasta los 550. El aprobador acepta la operación.

5.7.3.7. Caso de prueba VerySmallAmountLowRiskRejected

Este caso de prueba es análogo al anterior, variando la respuesta del aprobador, el cual decide rechazar la petición de préstamo.

5.7.3.8. Caso de prueba VerySmallAmountHighRisk

El clientTrack de este caso de prueba coincide con el de los dos anteriores, ya que el cliente vuelve a solicitar 500 de préstamo. En esta ocasión, el asesor decide que el riesgo de la operación es alto, por lo que se acepta la petición directamente. El servicio del aprobador no es invocado en esta ocasión.

5.7.4. Pruebas manuales

5.7.4.1. Comprobar la corrección de la composición adaptada

Una vez adaptada la composición *LoanApproval*, es necesario comprobar que sigue siendo un documento XML bien formado y que se despliega correctamente en ActiveBPEL. Para ello:

1. Se abre la composición adaptada con cualquier visor de archivos XML. Se recomienda el empleo de un navegador web, pues si no es sintácticamente correcta nos indicará por qué es así, es decir, si se nos ha olvidado una etiqueta de cierre o hemos cometido cualquier otro error sintáctico.

2. Una vez comprobada su validez sintáctica, se procede a ejecutar la composición frente al conjunto de casos de prueba. Para ello, se emplea la propia herramienta **MuBPEL**:
   ```
   mubpel run LoanApprovalCoverage.bpts
   LoanApprovalCoverage.bpel > Salida.xml
   ```
   Luego, es necesario abrir el fichero `Salida.xml`, al igual que antes, y comprobar que los casos de prueba se han pasado con éxito. (Aparecerá PASSED al comienzo del mismo). Si hubo algún tipo de error al probar, aparecerá ERROR o FAILURE y será necesario verificar el problema.

Si el fichero `Salida.xml` está vacío tras la ejecución, es por un fallo de despliegue de la composición original. Para averiguar el porqué del mismo, pueden consultarse los logs disponibles en `~/AeBpelEngine/Deployment-logs` y `~/AeBpelEngine/Process-logs`. En https://neptuno.uca.es/redmine/wiki/sources-fm/RecomendacionesDepuracion hay disponibles más consejos a la hora de depurar una composición WS-BPEL.

5.7.4.2. Comprobar la calidad del conjunto de casos de prueba

Si queremos verificar que nuestro conjunto de casos de prueba mata a todos los mutantes no equivalentes, es necesario obtener los resultados de la comparación. Por ejemplo, para verificar que nuestro conjunto de casos de prueba cumple el criterio de cobertura de sentencias:

```
mubpel compare LoanApprovalCoverage.bpts LoanApprovalCoverage.bpel
Salida.xml CFA_*_*.bpel > Ejecucion_CFA
```

Como podemos apreciar en 5.5, en todas las filas tenemos un 1. Esto quiere decir que existe, al menos, un caso de prueba que mata al mutante especificado en cada fila. En este caso en particular, si aparece una fila rellena de 2 es debido a la sustitución de la actividad `<reply>`, que provoca un fallo de desplegado.

Listado 5.5: Resultados de análisis de cobertura de sentencias

```
1  resources/loanCSV/CFA_10_1.bpel 1 0 0 0 0 0 0 0
2  resources/loanCSV/CFA_11_1.bpel 0 0 1 0 0 0 0 0
3  resources/loanCSV/CFA_1_1.bpel 1 0 0 0 0 0 0 0
4  resources/loanCSV/CFA_12_1.bpel 0 0 1 0 0 0 0 0
5  resources/loanCSV/CFA_13_1.bpel 0 0 0 1 0 0 0 0
6  resources/loanCSV/CFA_14_1.bpel 0 0 0 1 0 0 0 0
7  resources/loanCSV/CFA_15_1.bpel 0 0 0 1 0 0 0 0
8  resources/loanCSV/CFA_16_1.bpel 0 0 0 1 0 0 0 0
9  resources/loanCSV/CFA_17_1.bpel 0 0 0 1 0 0 0 0
10 resources/loanCSV/CFA_18_1.bpel 0 0 0 0 0 1 0 0
11 resources/loanCSV/CFA_19_1.bpel 0 0 0 0 0 1 0 0
12 resources/loanCSV/CFA_20_1.bpel 0 0 0 0 0 1 0 0
13 resources/loanCSV/CFA_21_1.bpel 0 0 0 0 0 1 0 0
14 resources/loanCSV/CFA_2_1.bpel 1 0 0 0 0 0 0 0
15 resources/loanCSV/CFA_22_1.bpel 0 0 0 0 0 1 0 0
16 resources/loanCSV/CFA_23_1.bpel 0 0 0 0 0 1 0 0
17 resources/loanCSV/CFA_24_1.bpel 0 0 0 0 0 1 0 0
18 resources/loanCSV/CFA_25_1.bpel 0 0 0 0 0 1 0 0
19 resources/loanCSV/CFA_26_1.bpel 0 0 0 0 0 0 0 1
20 resources/loanCSV/CFA_27_1.bpel 0 0 0 0 0 0 0 1
21 resources/loanCSV/CFA_28_1.bpel 2 2 2 2 2 2 2 2
22 resources/loanCSV/CFA_3_1.bpel 1 0 0 0 0 0 0 0
23 resources/loanCSV/CFA_4_1.bpel 1 0 0 0 0 0 0 0
24 resources/loanCSV/CFA_5_1.bpel 1 0 0 0 0 0 0 0
25 resources/loanCSV/CFA_6_1.bpel 1 0 0 0 0 0 0 0
```

```
26 resources/loanCSV/CFA_7_1.bpel 1 0 0 0 0 0 0 0
27 resources/loanCSV/CFA_8_1.bpel 1 0 0 0 0 0 0 0
28 resources/loanCSV/CFA_9_1.bpel 1 0 0 0 0 0 0 0
```

Si tuviéramos alguna fila rellena entera de 0 puede significar dos cosas:

- Nuestro conjunto de casos de prueba no es capaz de matar a todos los mutantes no equivalentes, por lo que sería necesario añadir más casos de prueba.
- Estamos ante un mutante equivalente, por lo que ningún caso de prueba podrá matarlo. Por ejemplo:
  Original:
  ```
  if (i * i > 23) {
  ...
  }
  ```
  Mutante:
  ```
  if (abs(i * i) > 23) {
  ...
  }
  ```
  En este ejemplo podemos observar un mutante equivalente ya que todo número elevado al cuadrado es positivo, es decir, $abs(i \cdot i) = i \cdot i$. La única manera de detectar los mutantes equivalentes es "manualmente", ya que no existe un método automático para ello, debido al problema del oráculo.

6. Conclusiones y trabajo futuro

6.1. Resumen

A continuación se resumirán los detalles más revelantes de PFC.

Se ha definido un conjunto de operadores de mutación para el lenguaje WS-BPEL 2.0, dividido en operadores que aplican criterios de cobertura y en operadores que en ciertas ocasiones pueden servir para aplicar criterios de cobertura.. Además, se han estudiado sus equivalencias con los operadores de cobertura definidos para otros lenguajes como C, Ada o Fortran.

Una vez concluida su definición, se han implementado los operadores de cobertura (CFA, CDE, CCO y, CDE) y los operadores que sirven para aplicar criterios de cobertura en ocasiones especiales (EIU, EIN, EAP y EAN).

A continuación, se ha definido un conjunto de casos de prueba unitarias en lenguaje Java, empleando el framework JUnit. Estas pruebas unitarias tienen como objetivo comprobar que los operadores de mutación se han implementado correctamente.

Una vez terminadas las pruebas, se ha seleccionado una composición WS-BPEL a la que pueda aplicarse estos operadores, de manera que se distinga adecuadamente la aplicación de los criterios de cobertura. Dado que no se encontró ninguna donde esto ocurriese, se decidió adaptar la composición *LoanApproval* para ello.

Al terminar la adaptación de la composición LoanApproval, se creó un conjunto de casos de prueba que matase a todos los mutantes no equivalentes generados tras aplicar los operadores fruto de este PFC, es decir, un conjunto de casos de prueba que cumple los criterios de cobertura de sentencias, decisión, condición y decisión / condición.

6.2. Valoración

Este PFC se ha llevado a cabo para colaborar en las tareas de investigación del grupo UCASE. Dado que los resultados de este proyecto son necesarios para el avance de la investigación sobre la prueba de mutaciones, el tiempo invertido ha sido el adecuado.

6.2.1. Objetivos

Se han cumplido todos los objetivos de este PFC. Ahora, el grupo UCASE dispone de una nueva categoría de operadores de mutación, con la que se pueden hacer estudios específicos de cobertura.

Para llevar a cabo este trabajo, ha sido de gran ayuda la documentación generada en [40], la cual, junto a esta memoria, serán de gran ayuda para las personas que deseen implementar nuevos operadores de mutación para WS-BPEL.

6.2.2. Conocimientos adquiridos

La experiencia ha sido muy enriquecedora para el alumno, el cual ha adquirido conocimientos en:

- Prueba de mutaciones, sobre todo lo relacionado a los operadores de mutación.
- Criterios de cobertura de código.
- Servicios web.
- Lenguajes: WS-BPEL, XSLT, XPath, Java.
- Frameworks: JUnit, BPELUnit.
- Herramientas: Dia, Eclipse, Subversion, Meld, Apache Maven.

Además, el alumno ha adquirido experiencia a la hora de trabajar en equipo, y ha aprendido que:

- Es necesario emplear un SCV en proyectos de mediana complejidad, pues permite la actualización del código fuente de una manera sencilla, permitiendo la posibilidad de revertir cambios perjudiciales.
- La combinación entre herramientas como Apache Maven y Jenkins permite a los desarrolladores depurar el código fuente de manera eficiente, ya que cada vez que exista una nueva revisión del código fuente, Jenkins procede a su compilación a través de Apache Maven, avisando al desarrollador en caso de que la revisión no sea estable e indicando las causas.
- A la hora de realizar trabajos de investigación, es importante el contacto personal con los tutores y demás miembros del grupo, ya que la bibliografía existente en estos casos es muy escasa, por lo que se tornan cruciales las opiniones e ideas que los miembros del grupo ofrecen durante los seminarios.

6.3. Trabajo futuro

En la sección 2.1.5 se hacía referencia a un estudio sobre la necesidad de un operador de cobertura múltiple para el lenguaje WS-BPEL. Los resultados obtenidos apuntaban a que podría no ser necesario este operador, dadas las características del lenguaje XPath en cuanto a las expresiones lógicas.

Sin embargo, estos resultados no son concluyentes, por lo que queda realizar un estudio más profundo acerca de esta cuestión.

Otra mejora de trabajo futuro consiste en la implementación del operador CDM [12]. Este operador aplicaría el criterio de cobertura de decisión / condición modificado, el cual busca mostrar el valor de cada condición independientemente de la decisión en la que esté presente. Esto hace que este criterio posea la mayoría de beneficios que acarrea el criterio de cobertura múltiple, con la ventaja de que el crecimiento del número de mutantes generados se mantenga lineal, en contra del crecimiento exponencial al aplicar el criterio de cobertura múltiple.

7. Agradecimientos

- A mi familia y amigos, por apoyarme y animarme en estos meses de trabajo y esfuerzo.
- A Inmaculada Medina Bulo, por darme la oportunidad de colaborar en el grupo UCASE mediante este proyecto.
- A Juan Boubeta Puig y Antonia Estero Botaro, por su dedicación y apoyo.
- A Antonio García Domínguez, por su ayuda prestada.
- Al resto de miembros del grupo UCASE, por su ayuda e ideas a la hora de resolver dudas.

A. Manual de instalación

En este capítulo se detalla cómo instalar la herramienta MuBPEL sobre una distribución GNU/Linux. En particular, se recomienda instalar la herramienta sobre *Ubuntu* u *openSUSE*, en su versión más reciente.

Instrucciones de instalación

Para instalar MuBPEL, es necesario:

1. Descargar el *script* de instalación en el directorio que se desee desde la URL: `https://neptuno.uca.es/redmine/projects/sources-fm/repository/changes/trunk/scripts/install.sh`
2. Ejecutar el script de instalación mediante la terminal, como sigue: `bash install.sh gamera`
3. Seguir las instrucciones que indique el script para concluir la instalación.

El *script* se encargará de instalar la herramienta MuBPEL y las dependencias de ésta:

- Apache Maven 3.0
- Java 6
- Apache Tomcat 5.5
- ActiveBPEL 4.1
- XMLBeans 2.3
- BPELUnit 1.5
- Saxon-B 9.1.0.1
- galib 2.4.7
- JUnit 4.8

B. Manual de usuario

A continuación ilustraremos el uso de la herramienta **MuBPEL**. Para entender la utilidad que posee la herramienta, es importante que el usuario posea conocimientos acerca de las prueba de mutaciones.

Como paso previo al uso de **MuBPEL**, es necesario instalar GAmera, mediante las instrucciones que se proveen en A.

Para ejecutar el programa se introduce en cualquier terminal el comando siguiente:

```
mubpel (argumentos)
```

Los argumentos serán explicados en las siguientes secciones.

B.1. Analizar una composición WS-BPEL

Mediante esta opción, se localizan las instrucciones del programa original que pueden ser mutadas. La sintaxis es:

```
mubpel analyze bpel
```

Se obtiene como resultado de esta operación un listado con los operadores disponibles, indicando por cada uno de ellos el número de localizaciones donde puede ser aplicado, y el valor máximo del atributo de cada operador.

Análisis de la composición *marketplace*

Si queremos analizar la composición *marketplace*, introduciremos en la terminal:

```
mubpel analyze marketplace.bpel
```

Obtendremos como salida:

Listado B.1: Salida del análisis de la composición marketplace

```
1  ISV 0 1
2  EAA 0 4
3  EEU 0 1
4  ERR 1 5
5  ELL 0 1
6  ECC 0 1
7  ECN 0 4
8  EMD 0 2
9  EMF 0 1
10 ACI 2 1
11 AFP 0 1
12 ASF 1 1
13 AIS 0 1
14 AIE 1 1
15 AWR 0 1
16 AJC 0 1
17 ASI 6 1
18 APM 0 1
19 APA 0 1
20 XMF 0 1
21 XMC 0 1
22 XMT 0 1
23 XTF 0 1
24 XER 0 1
25 XEE 0 1
26 AEL 6 1
27 EIU 0 1
28 EIN 1 1
29 EAP 0 1
30 EAN 0 1
31 CFA 6 1
32 CDE 1 2
33 CCO 1 2
34 CDC 1 2
```

Vemos que el operador CDE, por ejemplo, puede ser aplicado en 1 instrucción distinta dentro de la composición, ya que el valor de la localización es 1. Este operador realiza dos tipos de cambio, puesto que el valor máximo del atributo es 2, es decir, existe una única decisión en la composición, la cual puede ser sustituida por sus valores posibles.

B.2. Aplicar un operador a la composición original

Mediante esta opción, se aplica a la composición original el operador dado, generando un mutante. La sintaxis es:

```
mubpel apply bpel operator operand attribute
```

Es necesario, como puede apreciarse, indicar el operador, operando y valor del atributo para ello.

Se obtiene como resultado el mutante obtenido tras aplicar el operador en la localización dada con el atributo especificado en la salida estándar. Para su posterior uso, se recomienda redirigir ésta hacia un fichero.

Aplicar el operador CDE a la composición marketplace

El resultado del análisis para el operador CDE fue:

```
CDE 1 2
```

Así pues, podemos obtener 2 mutantes distintos. En el ejemplo, mutaremos la decisión de la composición por su valor verdadero, aplicando el operador como sigue:

```
mubpel apply marketplace.bpel CDE 1 1 > CDE_01_1.bpel
```

Como podemos ver, hemos guardado el mutante redirigiendo la salida estándar al fichero CDE_01_1.bpel. Para ver el cambio sintáctico realizado, podemos compararlo manualmente abriendo ambos archivos o emplear un visor de diferencias como Meld, ejecutando:

```
meld marketplace.bpel CDE_01_1.bpel &
```

Como podemos ver en la figura B.1, se ha generado el mutante que deseábamos.

B.3. Aplicar todos los operadores disponibles a la composición WS-BPEL original

Mediante esta opción, podemos aplicar todos los operadores analizados como aplicables a la composicióon, generando todos los mutantes posibles. La sintaxis es:

```
mubpel applyall bpel
```

Se obtienen tantos ficheros como mutantes posibles existan, de la forma *mAA-BB-CC.bpel*, donde *AA* representa el operador aplicado (entre 0 y el número de operadores menos 1), *BB* representa el operando y *CC* el valor del atributo.

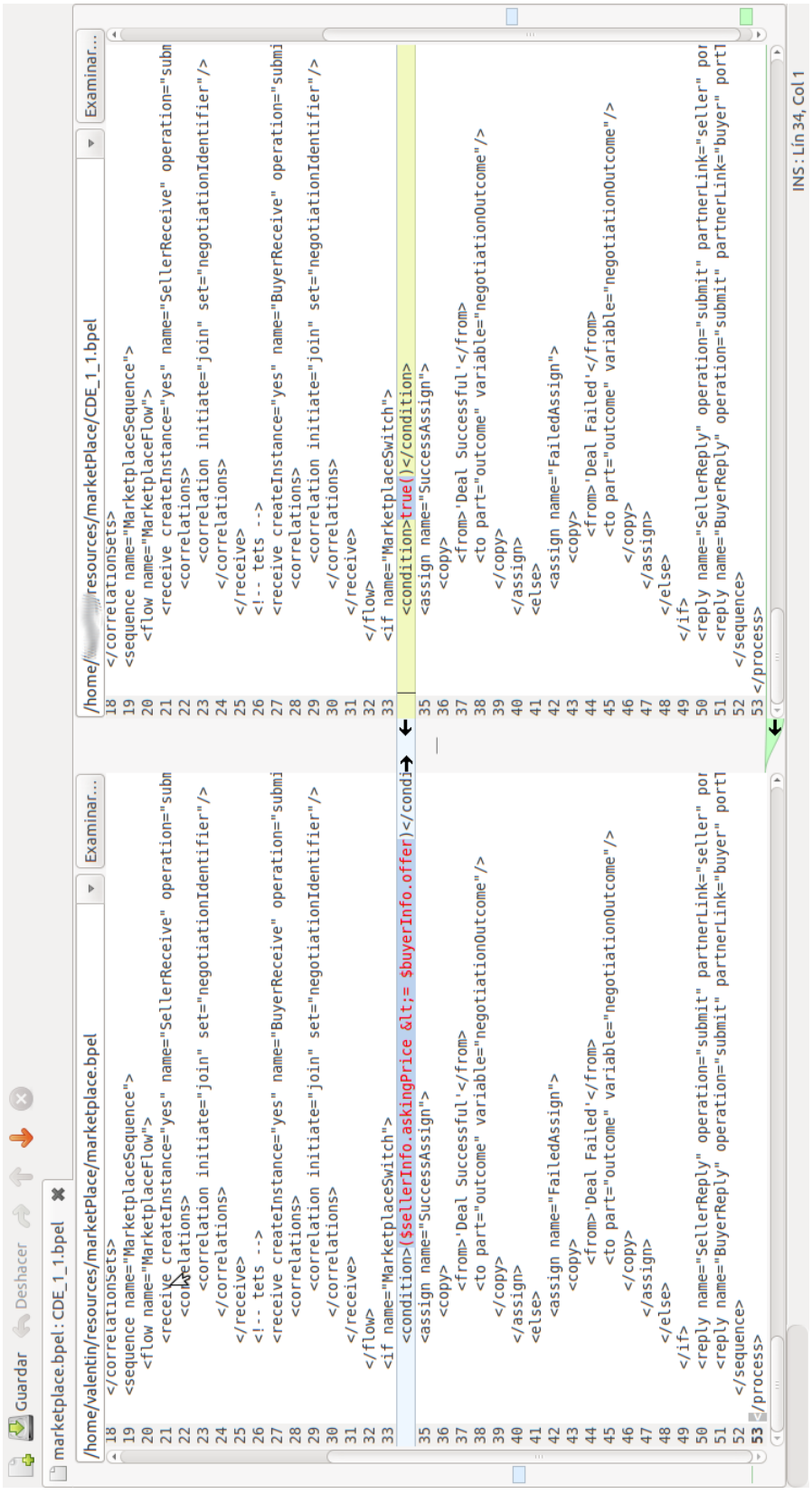

Figura B.1.: Diferencias entre la composición original y el mutante generado

Aplicar todos los operadores disponibles a la composición *marketplace*

Si queremos aplicar todos los operadores disponibles a la composición *marketplace*, introduciremos en la terminal:

```
mubpel applyall marketplace.bpel
```

Si queremos ver el conjunto de mutantes obtenido, ejecutaremos:

```
ls m*-*-*.bpel
```

Y veremos el conjunto de mutantes, cuyos nombre siguen la nomenclatura comentada anteriormente.

Listado B.2: Conjunto de mutantes de la composición marketplace

```
1  m04-01-01.bpel
2  m04-01-02.bpel
3  m04-01-03.bpel
4  m04-01-04.bpel
5  m04-01-05.bpel
6  m10-01-01.bpel
7  m10-02-01.bpel
8  m12-01-01.bpel
9  m14-01-01.bpel
10 m17-01-01.bpel
11 m17-02-01.bpel
12 m17-03-01.bpel
13 m17-04-01.bpel
14 m17-05-01.bpel
15 m17-06-01.bpel
16 m26-01-01.bpel
17 m26-02-01.bpel
18 m26-03-01.bpel
19 m26-04-01.bpel
20 m26-05-01.bpel
21 m26-06-01.bpel
22 m28-01-01.bpel
23 m31-01-01.bpel
24 m31-02-01.bpel
25 m31-03-01.bpel
26 m31-04-01.bpel
27 m31-05-01.bpel
28 m31-06-01.bpel
29 m32-01-01.bpel
30 m32-01-02.bpel
31 m33-01-01.bpel
32 m33-01-02.bpel
33 m34-01-01.bpel
34 m34-01-02.bpel
```

Una vez terminado este proceso, se obtendrán los ficheros con los mutantes, nombrados como se ha comentado anteriormente.

B.4. Ejecutar la composición original frente al conjunto de casos de prueba

Mediante esta opción, podremos ejecutar la composición original[1] frente al conjunto de casos de prueba para comprobar que se pasan con éxito. La sintaxis es:

```
mubpel run bpts bpel > Salida.xml
```

Además de la composición original, necesitamos el fichero *bpts* con los casos de prueba para ejecutar la composición. Se recomienda redirigir la salida estándar para obtener el resultado de la ejecución, pues es necesario para opciones posteriores.

Ejecutar la composición *marketplace* frente al conjunto de casos de prueba

Si queremos ejecutar la composición marketplace frente a su conjunto de casos de prueba, introduciremos en la terminal:

```
mubpel run marketplace.bpts marketplace.bpel > Salida.xml
```

Obtendremos el resultado de la ejecución de la misma en formato xml. Para visualizar la salida, es recomendable emplear un visor de XML como *xmleye*, que además, nos indicará visualmente si ha tenido éxito la ejecución. Como podemos ver en B.3, la ejecución de la composición marketplace ha tenido éxito, ya que se han pasado todos los casos de prueba (En la línea 2 podemos apreciar el resultado de la ejecución).

Listado B.3: Salida de la ejecución de la composición marketplace

```
1 ï»¿<?xml version="1.0" encoding="UTF-8"?>
2 <tes:testResult xmlns:tes="http://www.bpelunit.org/schema/testResult" name="Test
      Suite MarketPlace" result="PASSED" message="Passed">
3   <tes:state name="Status Code">PASSED</tes:state>
4   <tes:state name="Status Message">Passed</tes:state>
5   <tes:testCase name="Test Case SuccessfulDeal (Round 1)" result="PASSED" message
        ="Passed">
6     <tes:state name="Status Code">PASSED</tes:state>
7     <tes:state name="Status Message">Passed</tes:state>
8     <tes:partnerTrack name="Partner Track client" result="PASSED" message="Passed"
          >
```

[1] No es recomendable emplear esta opción para ejecutar un mutante, ya que se ejecuta el conjunto de casos de prueba al completo. Es mejor emplear la opción de comparación, que se explicará más adelante.

```
9       <tes:state name="Status Code">PASSED</tes:state>
10      <tes:state name="Status Message">Passed</tes:state>
11      <tes:activity type="SendReceiveSync" name="Send/Receive Synchronous" result=
            "PASSED" message="Passed">
12        <tes:state name="Status Code">PASSED</tes:state>
13        <tes:state name="Status Message">Passed</tes:state>
14        <tes:dataPackage name="Send Data Package" result="PASSED" message="Passed"
              >
15          ...
16        </tes:dataPackage>
17        <tes:dataPackage name="Receive Data Package" result="PASSED" message="
              Passed">
18          ...
19        </tes:dataPackage>
20      </tes:activity>
21    </tes:partnerTrack>
22    <tes:partnerTrack name="Partner Track Buyer" result="PASSED" message="Passed">
23      <tes:state name="Status Code">PASSED</tes:state>
24      <tes:state name="Status Message">Passed</tes:state>
25      <tes:activity type="SendReceiveSync" name="Send/Receive Synchronous" result=
            "PASSED" message="Passed">
26        <tes:state name="Status Code">PASSED</tes:state>
27        <tes:state name="Status Message">Passed</tes:state>
28        <tes:dataPackage name="Send Data Package" result="PASSED" message="Passed"
              >
29          ...
30        </tes:dataPackage>
31        <tes:dataPackage name="Receive Data Package" result="PASSED" message="
              Passed">
32          ...
33        </tes:dataPackage>
34      </tes:activity>
35    </tes:partnerTrack>
36  </tes:testCase>
37 </tes:testResult>
```

Así pues, en cada elemento `tes:testCase`, podemos apreciar si se tuvo éxito o no, para así identificar el caso de prueba que no se supera en el caso de una ejecución fallida.

B.5. Comparar la salida de la composición y la de los mutantes hasta la primera diferencia

Mediante esta opción, se compara la salida de la ejecución con la de los mutantes hasta encontrar la primera diferencia entre ellas. Para ello, previamente se ejecutan los mutantes, obteniendo respectivamente sus salidas. La sintaxis es:

```
mubpel compare bpts bpel xml (bpel-1...|-)
```

Se necesita pues, además de la composición original, el fichero de casos de prueba bpts, la salida de la ejecución de la composición original y los mutantes generados, separados por espacios. Se recomienda redirigir la salida estándar, como en opciones anteriores, para conservar los resultados de comparación.

Se obtiene como salida una fila por mutante, la cual comienza por el propio mutante, seguido de una serie de números, tantos como casos de prueba existan en el fichero *bpts* 5.6.1. Entonces:

- Si el número es 0, coinciden las salidas de la composición original y del mutante en ese caso de prueba.
- Si el número es 1, difieren las salidas en ese caso de prueba.
- Si el número es 2, estamos ante un mutante que no puede ser desplegado correctamente (No puede ser ejecutado).

Dado que se compara hasta la primera diferencia, la comparación se detendrá en el primer 1 de la fila y el resto de casos de prueba estará a 0. Si estamos ante un mutante inválido, la fila entera estará a 2.

Comparar la salida de la composición *marketplace* y la de los mutantes hasta la primera diferencia

Si queremos comparar la salida de la composición marketplace y la de los mutantes hasta la primera diferencia, introduciremos en la terminal:

```
mubpel compare marketplace.bpts marketplace.bpel Salida.xml CDE_-
01_1.bpel
```

Obtendremos como resultado lo siguiente:

```
CDE_01_1.bpel 0 0 1 0 0 0
```

Como podemos observar, el mutante dado ha muerto por el segundo[2] caso de prueba del fichero *marketplace.bpts*, es decir, porel caso de prueba de nombre *FailedDeal*. Debido a que estamos comparando hasta la primera diferencia, se termina ahí la comparación.

El resultado obtenido es el esperado, ya que en el mutante *CDE_01_1.bpel* se había sustituido el valor de la condición de la actividad *if* denominado `MarketplaceSwitch`,

[2]En marketplace.bpts, existen únicamente 3 casos de prueba. Sin embargo, en la salida aparecen 6. Esto es debido a que cada caso de prueba se ejecuta 2 veces, variando entre sí el tiempo de respuesta del cliente y del comprador. Por tanto, en este caso en concreto, el primer caso de prueba es la primera pareja, el segundo, la segunda, y así, sucesivamente.

haciéndola verdadera por defecto. En dicho caso de prueba, dicha condición no es verdadera, por lo que la salida difiere.

B.6. Comparar la salida de la composición y la de los mutantes

Mediante esta opción, se compara la salida de la ejecución con la de los mutantes, para todos los casos de prueba existentes. La sintaxis es:

```
mubpel comparefull bpts bpel xml (bpel-1...|-)
```

Esta opción es análoga a la opción *compare* B.5, pero en ésta la comparación es completa, es decir, se comparan las salidas en todos los casos de prueba, independientemente de si ha habido ya diferencias o no. Por ello, esta opción requiere de un mayor tiempo de ejecución.

Comparar la salida de la composición *marketplace* y la de los mutantes

Si queremos comparar la salida de la composición marketplace y la de los mutantes hasta la primera diferencia, introduciremos en la terminal:

```
mubpel comparefull marketplace.bpts marketplace.bpel Salida.xml
CDE_01_1.bpel
```

Obtendremos como resultado lo siguiente:

```
CDE_01_1.bpel 0 0 1 1 0 0
```

En este ejemplo, vemos que el caso de prueba número 2 es el único que mata al mutante. Como hemos seleccionado la comparación completa, no se ha detenido al encontrar la primera diferencia.

Como es obvio, la salida es correcta, ya que el segundo caso de prueba es el único en el que la condición toma el valor falso, y en el mutante, hacemos esa condición verdadera por defecto, como se reseñaba en la sección anterior.

B.7. Comparar dos salidas de ejecución de una composición

Mediante esta opción, se comparan dos salidas de la ejecución de una composición WS-BPEL, original o mutada, para visualizar variaciones en el comportamiento de algún caso de prueba, por ejemplo. La sintaxis es:

```
mubpel compareout xml1 xml2
```

Es necesario tener en cuenta de que el número de casos de prueba debe ser el mismo en ambas salidas.

B.8. Normalizar una composición

Mediante esta opción, se genera una forma canónica de una composición WS-BPEL. La sintaxis es:

```
mubpel normalize bpel
```

Redirigiendo la salida estándar hacia un fichero, podremos emplear la composición normalizada posteriormente. Se recomienda normalizar la composición a la hora de encontrar diferencias mediante un visor, o manualmente.

C. Manual del desarrollador

A continuación, se indican los pasos a seguir para desarrollar nuevos operadores de mutación para WS-BPEL 2.0.

C.1. Descarga del proyecto MuBPEL

Para comenzar con el desarrollo, es necesario descargar la última versión del proyecto de la forja *Redmine*. Para ello, se introduce el siguiente comando en la terminal[1]:

```
svn co https://neptuno.uca.es/svn/sources-fm/trunk/src/mubpel
```

Una vez finalizado, se tendrá una copia de trabajo del repositorio en la que se podrá comenzar con el desarrollo.

C.2. Creación del proyecto para Eclipse

Una vez descargada la copia de trabajo, es necesario generar un proyecto para Eclipse, para facilitar la labor del desarrollo. Para ello, en el mismo directorio donde se descargó la copia, se ejecuta:

```
mvn eclipse:eclipse
```

Una vez realizado, se habrá generado un proyecto para Eclipse, el cual podrá ser importado en dicho IDE para comenzar a desarrollar.

En Eclipse será necesario crear la variable de entorno *M2_REPO*, la cual es necesaria para resolver las dependencias del proyecto en el repositorio *maven*. Para ello, se acude a *Window* → *Preferences*, y allí a *Java* → *Build Path* → *Classpath Variables* (Ver figura C.1).

Se crea una nueva variable, con el nombre dado anteriormente, cuyo valor será el directorio donde se aloja el repositorio maven (normalmente `~/.m2/repository`).

Una vez está el entorno listo, se podrá comenzar a trabajar.

[1] Es necesario tener instalado el sistema de control de versiones Subversion en la máquina.

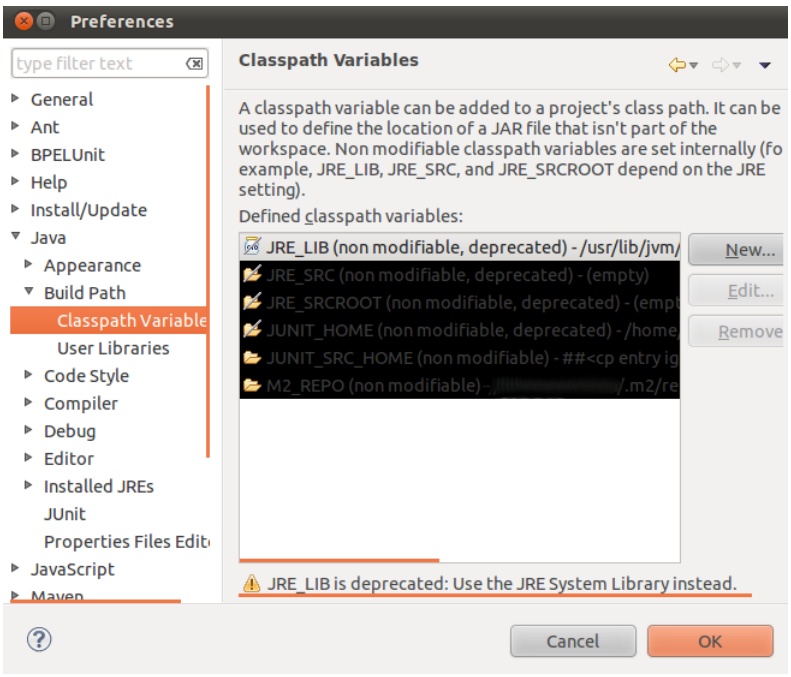

Figura C.1.: Variable M2_REPO en Eclipse

C.3. Implementación de operadores de mutación para WS-BPEL 2.0

La metodología de trabajo para desarrollar nuevos operadores de mutación para WS-BPEL 2.0 es la que sigue a continuación:

1. Se implementa el código XSLT que realiza la transformación deseada a la composición original WS-BPEL. Atendiendo al tipo de mutación que realice 5.5.3.2:

   a) Si elimina actividades del proceso, heredará de `delete-op-base.xsl`.

   b) Si afecta a expresiones XPath, heredará de `xpath-op-base.xsl`.

   c) Si afecta a otro tipo de actividades, heredará de `op-base.xsl`.

   Esta hoja XSLT deberá de guardarse en el directorio `mubpel/src/main/resources/es/uca/webservices/mutants/operators`.

2. Se añade en el vector *OPERATOR_NAMES* del fichero `operatorConstants.java` el nombre del nuevo operador, para que pueda crearse la instancia del nuevo operador en la fábrica. Este fichero está disponible en `mubpel/src/main/java/uca/webservices/mutants/operators`.

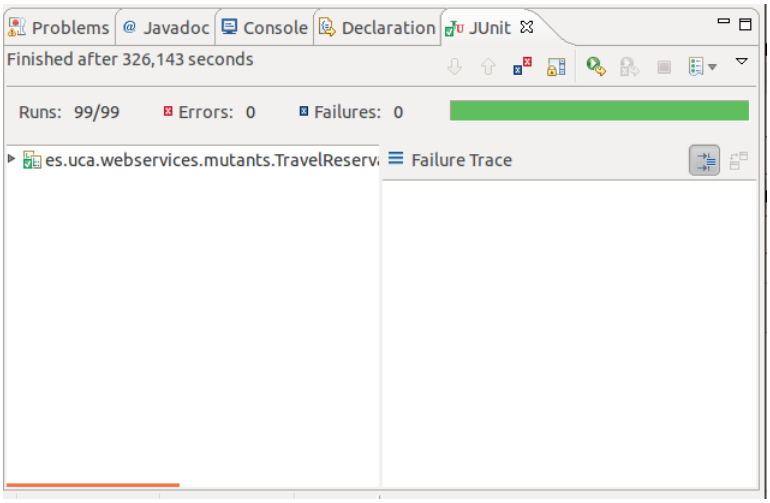

Figura C.2.: Ejecución de las pruebas unitarias en eclipse

3. Se implementan las pruebas unitarias empleando JUnit, en los ficheros de prueba cuyo nombre coincide con el de la composición en la que va a ser probado el nuevo operador, dentro del directorio `mubpel/src/test/java/uca/webservices/mutants`.

   Los casos de prueba definidos deben verificar:

   a) El número de operandos donde se puede aplicar el operador.

   b) El cambio sintáctico que realiza el operador.

   c) La diferencia de salida entre la composición original y el mutante.

En el caso de que el operador desarrollado no sea aplicable en las composiciones WS-BPEL existentes, deberá crear una nueva o adaptar una existente, para que así el nuevo operador pueda ser aplicado. En ambos casos, además, deberá generar casos de prueba en el fichero *bpts* para que mueran todos los mutantes no equivalentes que haya generado el nuevo operador.

C.4. Ejecución de las pruebas unitarias

Una vez implementados los nuevos operadores, es necesario ejecutar las pruebas unitarias para verificar su correcto funcionamiento. Para ello existen dos opciones:

1. Desde Eclipse, se abre el fichero de casos de prueba, y se acude a Run → Run As → JUnit Test. El entorno procederá a ejecutar el conjunto de casos de prueba dado, indicando al finalizar si se han pasado con éxito. En el caso de haber algún caso de prueba no superado, se indica el motivo. (Ver figura C.2)

De esta forma, se pueden ejecutar las pruebas individualmente, es decir, las pruebas del operador para una composición en concreto. Esto es más cómodo para el proceso de depuración del operador.

2. Desde el directorio `mubpel`, se ejecuta:

   ```
   mvn test
   ```

   De esta forma, se ejecutan todos las pruebas unitarias, y se genera un informe detallado. Esto es más cómodo para verificar que todo funciona correctamente, una vez se finaliza el desarrollo de los operadores.

C.5. Generación del ejecutable

Una vez concluido el proceso de prueba, se puede generar una nueva versión de **MuBPEL**, la cual contendrá los nuevos operadores implementados. Para ello, será necesario generar un nuevo *.jar*, el cual se obtiene mediante la orden:

```
mvn package
```

De esta forma se obtiene un fichero en el directorio `mubpel/target` cuyo nombre tiene la siguiente forma: *mubpel-x.y.z-SNAPSHOT-dist.tar.gz*.

Descomprimiendo ese fichero en el directorio `~/bin` y sustituyendo la versión anterior, puede ejecutarse la nueva versión de **MuBPEL**:

```
mubpel (argumentos)
```

Bibliografía

[1] *BPELUnit - The Open Source Unit Testing Framework for BPEL*, Septiembre 2011. URL http://bpelunit.net/.

[2] *Meld Home Page*, Septiembre 2011. URL http://meld.sourceforge.net/.

[3] *Página del proyecto RapidSVN*, Agosto 2011. URL http://rapidsvn.tigris.org/.

[4] *Wiki del proyecto ActiveBPEL*, Septiembre 2011. URL https://neptuno.uca.es/redmine/projects/activebpel.

[5] *Wiki del proyecto MuBPEL*, Septiembre 2011. URL https://neptuno.uca.es/redmine/projects/sources-fm/wiki/MuBPEL.

[6] *Xacobeo Home Page*, Septiembre 2011. URL https://code.google.com/p/xacobeo/.

[7] Hiralal Agrawal, Richard A. DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, y Eugene Spafford. *Design of Mutant Operators for the C Programming Language*. Informe Técnico SERC-TR-41-P, 1989.

[8] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, y Jérôme Siméon. *XML Path Language (XPath) 2.0*. Informe técnico, 2010. URL http://www.w3.org/TR/xpath20/.

[9] Paul V. Biron y Ashok Malhotra. *XML Schema Part 2: Datatypes*. Informe técnico, 2004. URL http://www.w3.org/TR/xmlschema-2/.

[10] Juan Boubeta-Puig, Inmaculada Medina-Bulo, y Antonio García-Dominguez. *Analogies and Differences between Mutation Operators for WS-BPEL 2.0 and Other Languages*. En *6th Workshop on Mutation Analysis*. Pendiente de publicación, Berlín, Alemania, 2011.

[11] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, y François Yergeau. *Extensible Markup Language (XML) 1.0*. Informe técnico, 2008. URL http://www.w3.org/TR/xml/.

[12] J. J. Chilenski y S. P. Miller. *Applicability of modified condition/decision coverage to software testing*. *Software Engineering Journal*, 9(5):193–200, sep 1994. ISSN 0268-6961.

[13] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, y Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. Informe técnico, 2007. URL http://www.w3.org/TR/wsdl20.

[14] Jenkins CI. *Jenkins Home Page*, Septiembre 2011. URL https://jenkins-ci.org.

[15] Ben Collins-Sussman, Brian W. Fitzpatrick, y C. Michael Pilato. *Version Control with Subversion*. OReilly Media, jun 2004. Http://svnbook.red-bean.com/nightly/en/index.html.

[16] Edsger W. Dijkstra. *Chapter I: Notes on structured programming*. páginas 1–82, 1972.

[17] Antonio García Domínguez. *XMLEye Home Page*, Septiembre 2011. URL https://neptuno.uca.es/redmine/projects/sources-fm/wiki/XMLEye.

[18] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, y I. Medina-Bulo. *Evolutionary Mutation Testing*. *Information and Software Technology*, 53(10):1108–1123, 2011.

[19] Juan José Domínguez-Jiménez, Antonia Estero-Botaro, Antonio García-Domínguez, y Inmaculada Medina-Bulo. *GAmera: An Automatic Mutant Generation System for WS-BPEL Compositions*. En *ECOWS 2009: Seventh IEEE European Conference on Web Services*, páginas 97–106. IEEE Computer Society, Eindhoven, The Netherlands, 2009.

[20] Juan José Domínguez-Jiménez, Antonia Estero-Botaro, y Inmaculada Medina-Bulo. *A Framework for Mutant Genetic Generation for WS-BPEL*. En *SOFSEM 2009: Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science, Lecture Notes in Computer Science*, tomo 5404, páginas 229–240. Springer-Verlag, 2009. ISBN 978-3-540-95890-1.

[21] Antonia Estero-Botaro, Francisco Palomo-Lozano, y Inmaculada Medina-Bulo. *Mutation Operators for WS-BPEL 2.0*. En *21th International Conference on Software & Systems Engineering and their Applications*. Paris, Francia, 2008.

[22] The Apache Software Foundation. *Apache Maven Home Page*, Septiembre 2011. URL https://maven.apache.org/.

[23] The Apache Software Foundation. *Apache Tomcat Home Page*, Septiembre 2011. URL https://tomcat.apache.org/index.html.

[24] The Apache Software Foundation. *XMLBeans Home Page*, Septiembre 2011. URL https://xmlbeans.apache.org/.

[25] The Eclipse Foundation. *Eclipse Home Page*, Septiembre 2011. URL https://www.eclipse.org/.

[26] Martin Fowler. *Continous Integration*. 2006. URL http://martinfowler.com/articles/continuousIntegration.html.

[27] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Patrones de Diseño*. Addison-Wesley, 2003.

[28] Y. Jia y M. Harman. *An Analysis and Survey of the Development of Mutation Testing*. *Software Engineering, IEEE Transactions on*, PP(99):1, 2011. ISSN 0098-5589. doi:10.1109/TSE.2010.62.

[29] Juan José Domínguez Jiménez, Antonia Estero Botaro, y Inmaculada Medina Bulo. *Generación de mutantes con algoritmos genéticos*. En *Actas del III Taller sobre Pruebas en Ingeniería del Software*, tomo 2, páginas 27–32. 2008.

[30] Michael Kay. *Saxon Home Page*, Septiembre 2011. URL http://saxon.sourceforge.net/#F9.3HE.

[31] K. N. King y A. Jefferson Offutt. *A FORTRAN Language System for Mutation-based Software Testing*. *Software – Practice and Experience*, 21(7):685–718, 1991.

[32] Yu-Seung Ma, Jeff Offutt, y Yong-Rae Kwon. *MuJava: An Automated Class Mutation System*. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[33] Sun Microsystems. *NetBeans*, diciembre 2009. Http://www.netbeans.org/.

[34] MIT. *GAlib Home Page*, Septiembre 2011. URL http://lancet.mit.edu/ga/.

[35] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, segunda edición, 2004.

[36] OASIS. *Web Services Business Process Execution Language 2.0*, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[37] A. Jefferson Offutt, Jeff Voas, y Jeff Payne. *Mutation Operators for Ada*. Informe Técnico ISSE-TR-96-09, 1996.

[38] Oracle. *Java 6 overview*, Septiembre 2011. URL http://www.oracle.com/technetwork/java/javase/overview/index-jsp-136246.html.

[39] JUnit. org. *Resources for Test Driven Development*, ago 2011. URL http://www.junit.org/.

[40] Juan Boubeta Puig. *Implementación de operadores para WS-BPEL 2.0*. 2010. URL http://rodin.uca.es:8081/xmlui/handle/10498/9755.

[41] Ian Sommerville. *Ingeniería del Software: Un Enfoque Práctico*. Pearson, séptima edición, 2005.

[42] Javier Tuya, María J. Suárez-Cabal, y Claudio de la Riva. *Mutating database queries*. *Information and Software Technology*, 49(4):398–417, 2007.